MIT/LCS/TR-250

FAULT TOLERANCE IN PACKET COMMUNICATION

COMPUTER ARCHITECTURES

Clement Kin Cho Leung

This blank page was inserted to preserve pagination.

Fault Tolerance
in
Packet Communication Computer Architectures

by

Clement Kin Cho Leung

September 1980

This research was supported by National Science Foundation
under grant MCS 75-04060 A01

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge Massachusetts 02139

Fault Tolerance in Packet Communication Computer Architectures

by

Clement Kin Cho Leung

Submitted to the Department of Electrical Engineering and Computer Science on 28 August 1980
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

It is attractive to implement a large scale parallel processing system as a self-timed hardware system with decentralized control and to improve maintainability and availability in such a system through fault tolerance. In this thesis we show how to tolerate hardware failures in a self-timed hardware system with a packet communication architecture, designed to execute parallel programs organized by data flow concepts.

We first formulate a design methodology for incorporating redundant hardware into self-timed systems for fault tolerance. Redundancy management problems in self-timed systems are illustrated with a byte-sliced hardware module structure. Robust algorithms are given for synchronizing byte slices in a redundant module so that their outputs can be decoded to detect and/or mask hardware failures. Hardware implementation of these redundancy management algorithms is studied under a stuck-at fault model, a random pulse train fault model and a random wave train fault model.

In studying the design of fault-tolerant data flow processors we have also developed a dynamic redundancy scheme for masking hardware failures in a multiprocessor architecture designed to execute parallel programs organized by data flow principles. Novel features of this architecture include use of packet networks to support communication among processing elements and dynamic allocation of a homogeneous set of functional units to service requests. Program organization and hardware module designs to support the dynamic redundancy scheme are described.

Thesis Supervisor : Jack B. Dennis

Title : Professor of Electrical Engineering and Computer Science

Keywords : Data flow computer architecture, self-timed hardware systems, fault tolerance, dynamic redundancy, fault-tolerant networks, fault-tolerant synchronization, non-determinacy.

Acknowledgments

I would first of all like to express my gratitude to my thesis supervisor, Professor Jack B. Dennis, who heads the Computation Structures Group at the Laboratory for Computer Science at MIT. He has been a constant source of encouragement and insight during my graduate studies. He and Professor Michael L. Dertouzos, director of the Laboratory for Computer Science at MIT, have also provided me with numerous opportunities for professional advancement. From them I have learned a great deal.

I thank my readers, Professors Elias and Halstead, for their many constructive comments. Critical comments from Dr. Liba Svobodova, now of IRIA, on an earlier draft of this thesis are also gratefully acknowledged.

I am much indebted to the Department of Electrical Engineering and Computer Science at MIT which has provided me with financial support through teaching assistantships and instructor appointments.

I would also like to thank Professor Frederick C. Hennie III, my faculty advisor, for his guidance and support over the years.

Members and former members of the Computation Structures Group, in particular William B. Ackerman, Sheldon and Sandy Borkin, Andy Boughton, Dean and Ruth Brock, Randy Bryant, Lynn Montz and Ken Weng, have provided this foreign student and his wife a home away from home.

My eight years of graduate studies have been very rewarding academically. For other than academic reasons, these eight years have also been quite trying emotionally. Members of my family, my wife Enid by my side, and my parents and brothers from afar, have shared the frustrations of a seemingly endless graduate program with me, and have supported me selflessly.

Table of Contents

1. Introduction

In this thesis we study the following fault tolerance problem:

"How can hardware failures be tolerated in a self-timed hardware system which is organized by a packet communication architecture and designed to execute data flow programs ?"

We first explain our motivations for studying fault tolerance problems in this context.

Computer systems which are significantly more powerful than those presently available are needed in weather forecasting and aeronautical design. These physics problems are often formulated as a set of difference equations for numerical solution. It is readily apparent from studying these equations that many intermediate values required for their solution can be computed in parallel; and so it is attractive to design computer systems which can exploit such parallelism through hardware concurrency. Many approaches to organize a large number of hardware units so that a good portion of them can participate in the computation in progress simultaneously have been studied. Architectures based on vector processing and array processing are available commercially. Another approach which appears to be very promising, and provides the environment in which the research reported in this thesis is carried out, is to construct computer systems based on data flow concepts. Current projects on data-driven computations and highly concurrent systems based on these concepts are surveyed in [50], [51].

The data flow approach to parallel processing is based on the observation that an operation is enabled for execution as soon as the operands it needs are available, and hence need not be further delayed by other artificial sequencing constraints. In the internal representation of an application program in a data flow system, data dependencies among operations are stored explicitly and used

directly to schedule operations for execution. In the data flow computer architectures [24], [25] studied at MIT, application programs are stored as program graphs, and hardware units are designed to execute program graphs directly.

In a conventional computer organization, operation of hardware units such as arithmetic-logic units, shifters and register files are coordinated in machine instruction execution by control signals generated by a control unit. Many mainframe computers, minicomputers and microcomputers are implemented under this control strategy. We call this mode of hardware operation *centralized control.* To gain orders of magnitude improvement in performance over state-of-the-art uniprocessors, a parallel processing system must incorporate a large number of hardware units and exploit parallelism in computations to achieve high throughput. It seems that to operate a large number of hardware units under centralized control, excessive demand will be placed on the complexity and performance of the control unit. Performance is compromised if the control unit becomes a system bottleneck due to the complexity of the control tasks it must carry out. On the other hand, only restricted forms of parallelism, such as pipelining and array processing, can be realized as simple control tasks. For a large scale architecture it is thus attractive to use *decentralized control.* Packet communication architecture is an organizational scheme to support decentralized control.

Under the packet communication principle hardware units in a system communicate with each other by exchanging information packets, and packet processing is carried out at each unit autonomously. The size and complexity of these units depend on both system design and implementation technology. In an architecture designed for high speed numerical computation, for example, each unit may consist of a small number of VLSI chips and a packet may consist of a pair of floating point numbers and some control information.

The packet communication discipline has been used to organize several data flow processors [24], [26] and a high throughput memory system [21]. We have also studied design methodologies for constructing hardware systems organized by a packet communication architecture, which we shall refer to as *packet communication systems* [33]. Formal methods for studying properties of packet communication systems have been studied in [27] and [32]. In this thesis we study architectural organizations appropriate for tolerating hardware failures in packet communication computer architectures, and will illustrate our ideas with a packet communication architecture designed to execute a class of data flow programs. As part of this study we have also developed a set of concepts and techniques for incorporating redundant hardware into *self-timed hardware systems* [48] to combat hardware failures.

In a conventional computer organization low level hardware synchronization, between a combinational circuit and its input and output registers, for example, is governed by a set of periodic signals generated by a system clock. Due to the use of a system clock, these hardware systems are also called *synchronous* hardware systems. A decentralized control computer architecture can be implemented as a synchronous system. A large scale synchronous system, however, suffers from poor modularity because synchronous operation introduces strong interaction among hardware modules, indirectly through the timing characteristics of the clock signals. The design of each hardware module must be consistent with the period and frequency of the clock signals, even though these characteristics may have been imposed by the synchronization requirements of other hardware modules. When two neighboring hardware modules synchronize their input, processing and output activities indirectly by referencing a common timing signal and one of these modules is modified, the modification must not violate the timing assumptions made in the other module. Otherwise the other module must be modified accordingly. There are also skew problems associated with clock distribution in a large scale system. For these reasons we also favor implementing packet

communication system as an interconnection of hardware modules which exchange packets under asynchronous two-way handshake protocols.

Under an asynchronous packet communication protocol, packet communication between modules is synchronized entirely through exchanging signals among them. We call hardware modules which interact with their environment via these protocols *packet communication modules.* An example of asynchronous packet communication is given in Fig. 1.1. Packets are delivered as a sequence of packet bytes over an 8-bit *data link* (Fig. 1.1a). Packet byte receipt and transmission are synchronized by exchanging control signals between the sender and the receiver over the *control link.* Packet communication using the control discipline of *transition signaling* is illustrated in Fig. 1.1b.

Since a packet communication system makes use of handshake protocols for sending packets between modules, it is also a self-timed hardware system. The redundancy management concepts and techniques developed in this thesis have been tailored specifically for packet systems, but they are also applicable to other classes of self-timed hardware systems [4], [31], [39], [49].

Let us consider designing high performance computing systems for physics simulation such as in weather forecasting and aeronautical design. The physics simulation environment is different from traditional application domains for fault-tolerant computers, such as process control and interplanetary spaceflight, in that it inherently has no stringent reliability requirements. It is nonetheless attractive to use fault tolerance techniques to improve availability and maintainability. System throughput is improved if hardware failures can be masked, especially since it is not unusual for a numerical computation in physics simulation to execute for many hours. It is extremely annoying if the entire computing system is disabled because of one malfunctioning transistor or one loose connection among the hundreds of thousands in use. These considerations have motivated the research reported in this thesis on fault tolerance in packet communication computer architectures.

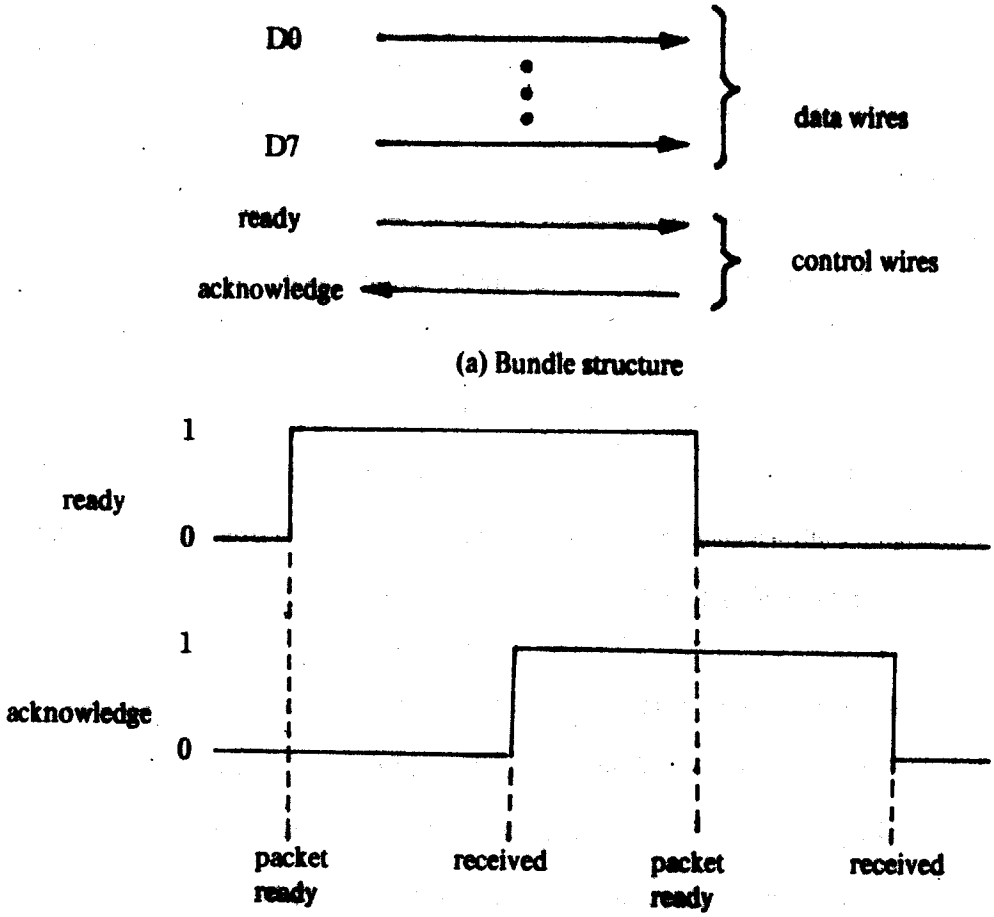


Fig. 1.1. An asynchronous packet communication protocol.

The problems studied in this thesis will be explained in more detail after we introduce some fundamental ideas for designing computing systems which can tolerate hardware failures.

1.1 Fault Tolerance

1.1.1 Basic Concepts

The following paragraphs on fault tolerance for physical faults is excerpted from the paper "Fault-tolerance: The survival attribute of digital systems" by Professor Algirdas Avizienis of UCLA, published in the Proceedings of the IEEE, vol. 66, no. 10. [53]

"The notion of fault tolerance for physical faults as an attribute of digital systems requires the viewing of the system from two different viewpoints: physical and logical. From the viewpoint of purely physical observation and measurement, digital systems are physical systems, built of large assemblies of electronic, electromechanical and other physical components. The components obey the laws of physics and the behavior of the system can be described in terms of time functions of the values of physical variables: voltages, currents, positions, etc.

"From the viewpoint of the human society at large digital systems are 'black boxes' that perform information processing to meet human needs. In the very general view of Newell and Simon [40], they serve as physical symbol systems, in which physical variables represent other entities - symbols, expressions, and processes, and the system is capable of designation and interpretation. At the level of greatest formal detail, the information processing behavior of a digital system may be specified in terms of a finite-state sequential machine. Outside of the formal interpretations, the daily user sees his box of digital hardware as a useful artifact that can store large quantities of symbolic information and can carry out information processing operations that are specified by his programs. In this perspective, it is only the logical activity of information processing that concerns the user, and not the physical properties of the circuits inside. As long as the logic machine runs well, what happens with the physical system that is its host is of very little, if any, interest.

"The use of the physical system as a logic machine is based on the conventions that the values of physical variables are interpreted as the discrete values of logic variables, and that the speed with which transformations are carried out is limited by the physical properties of the hardware. The logic machine behaves in the specified manner as long as the parameters of physical components and the speed of operation remain within specified limits. However, it has been a common experience that unexpected out-of-specification physical changes in component parameters do occur in all kinds of digital hardware. They are usually called *malfunctions* when the changes are temporary, and *failures* when the changes are permanent. Their effect is to cause an unspecified and disruptive change of one or more logic variables of the logic machine. Such a change of logic values is called a *physical fault*, or simply a *fault* when the physical cause of the fault is clear from the context of the discussion, as it is here. ...

"It is the possibility of randomly occurring faults that makes the user uncomfortably aware of the physical side of his logic machine. The occurrence of a fault often causes an *error*, that is, a deviation of the logic machine from its program-specified behavior (transition through a sequence of specified states) into a sequence of error states. While in an error state, the logic machine fails to perform correctly at least one of its specified tasks, and suffers a partial or complete failure to carry out the information processing function. Such failures are a commonplace experience for the users of contemporary information processing systems. The usual solution of the problem is a manually controlled maintenance action that results in the removal or repair of the cause of the fault. The system is then restarted to run until the next fault strikes.

"The purpose of fault-tolerance is to offer an alternate solution to the fault problem in which the detection of faults and the recovery to normal operation are carried out as internal functions of the system itself. *Fault-tolerance* is the unique attribute of a digital system which makes it possible for the system to continue with its program-specified behavior as a logic machine after the occurrence

of faults. It may be said that fault-tolerance is the *survival attribute* of the logic machine because its purpose is to cause a return from error states back to the specified behavior, thus assuring the survival of the information processing activities.

" ... fault-tolerance requires additional hardware and/or software that is *redundant* during normal operation and would be entirely superfluous in a completely fault-free logic machine. In the fault-prone physical implementation, fault-tolerance is the insurance of the logic machine against disruptive physical events. Its function is to react to the occurrence of faults and to protect the logic machine against the imperfections of the physical system that serves as its host. ... "

The above excerpt serves to introduce the basic notions of *failures, faults, errors and fault tolerance* for fault-tolerant computing. We use the terms failures, faults and errors as explained above. A *failure* is an unexpected out-of-specification physical change in component parameters. Examples of failures are out-of-spec transistors and loose connections. A *fault* is an unspecified and disruptive change of logic values caused by a failure. A failure occurs in a physical machine while a fault occurs in a logical machine. A fault model relates the two phenomena by postulating how logic values may change when physical failures occur. The most common fault model is the stuck-at fault model which postulates that logic variables will become stuck at either logical 1 or logical 0 as a result of physical failures. An *error* is a deviation of the logic machine from its program-specified behavior into a sequence of error states due to faults. Faults may be tolerated such that their occurrences do not cause errors. A simple example of fault tolerance is to use bit-sliced memory systems protected by error correcting codes so that when a single bit-slice fails and causes one bit of the data word to be faulty, the contents of the word can still be recovered and no error can occur.

We shall restrict our attention to tolerating physical failures in this thesis, and will not deal with human mistakes that occur when the computing system is designed, implemented or in use.

Fault-tolerant systems differ in the protections each offers against physical failures. For system design the related notions of maintainability and availability are also useful. Maintainability is the property of having low maintenance costs. Availability is a measure of the fraction of the useful life of a computing system during which the system is available for performing the functions it is designed for. Communication processors are often designed to be highly available, even though there is no guarantee that conversations in progress when failures occur will not be disrupted.

The logical machine we work with in this thesis is a computer system designed to execute a class of data flow programs. The underlying physical machine is a hardware system organized by a packet communication architecture. We study architectural organizations and hardware redundancy techniques appropriate for tolerating certain types of hardware failures in the physical machine. Our design goal is that a data flow program running on the computer system when failures in the specified class occur can always run to completion, and compute the same results as if no failure has occurred. In other words, the program execution capabilities of the logical machine are preserved as long as only failures in the specified class occur in the physical machine. The system design presented in this thesis achieves fault tolerance by integrating program organization, architectural features and redundant hardware in the physical machine into a system strategy. This strategy is effective only for certain types of hardware failures in the physical machine. Hardware failures in a physical machine are characterized by their *extent*, as well as by fault models. We will study fault models for packet communication systems in detail. For now we state our assumptions about the extent of hardware failures that can be tolerated by our strategy. Since the extent of failures can only be described in terms of physical structure, we next introduce the structure of redundant hardware modules used in this thesis.

1.1.2 Structure of Redundant Packet Communication Systems

We propose a design methodology for fault-tolerant packet communication systems consisting of the following steps:

(1) A packet communication system is designed according to its functional specification. We refer to this initial design as the non-redundant system.

(2) Fault tolerance is incorporated into the non-redundant system by incorporating redundant hardware into each module in the non-redundant system and into the connections between modules. The product of this second step is a redundant system topologically similar to the non-redundant system, but constructed out of redundant modules connected through redundant links.

For this thesis, we furthermore assume that redundant packet communication modules have a byte-sliced internal organization, constructed with failure-independent *byte slices*.

Byte slicing is a common technique for implementing memory systems and arithmetic-logic processors. A memory chip or an arithmetic-logic chip implements the specified function for fixed size bytes. An array of such chips is used to implement these functions for multiple-byte words. In general, each byte slice should receive all input words, and control and data are passed between slices in the same array. (In a memory system or an arithmetic-logic processor each byte-slice receives only one byte of each input word, and in a memory system there is no communication between memory chips.) An output word of a byte-sliced module consists of several bytes, one from each slice.

For fault tolerance, redundant slices are introduced into a byte-sliced module and the bytes in each output word are correlated by an *encoding scheme*. In an encoding scheme a hardware module is modeled as a function which maps an input bit vector into an output bit vector, and a module

failure is modeled as a *fault function* which maps a fault-free output bit vector into a faulty one. If each output bit vector consists of n bits, then an encoding scheme C partitions the set of all n-bit vectors into a code space S_C and a non-code space N_C. To detect hardware failures in a module, C is chosen such that:

(1) All output vectors generated by the module during fault-free operation are in S_C, and

(2) Under the most common hardware failure modes in the module no faulty bit vector in S_C can be generated. This property implies that under the most common failure modes the outputs generated by the module is either the fault-free result or is a non-codeword in N_C.

When the output words of a module are encoded by a scheme C satisfying these two properties, the most likely hardware failures in the module can be detected by checking these words for code space violations. An encoding scheme can be used to mask failures in a hardware module if it satisfies (1), (2) and (3):

(3) Under the most common hardware failure modes in the module, each faulty output vector in N_C can be decoded to derive the fault-free vector.

The byte-sliced organization is particularly amenable to applying encoding techniques since under some of the most common failure modes, failure of a single byte slice, for example, only one byte in each output word will be affected. Since the pioneering work of Hamming [29], much work has been done on applying coding techniques to detect and mask failures in byte-sliced arithmetic processors and memory systems [6], [13], [65], [47]. Hardware failures are also often dealt with using *modular redundancy*. In a modular redundancy scheme several copies of a hardware module are operated together and their outputs are compared or voted upon to detect and mask hardware failures among them. Modular redundancy can be regarded as a special case of byte slicing and output encoding in which all slices perform identical functions and each output word consists of two

or more identical bytes.

Each byte-sliced module M in a redundant packet communication module is designed so that failures among byte slices in it can be detected and/or masked by decoding their outputs, as long as the *extent* of hardware failures is limited to less than some fixed number, M_N, of byte slices in the module.

A redundant packet communication system constructed using these byte-sliced modules can only tolerate hardware failures whose extents are thus limited. It is a reliability improvement over its nonredundant counterpart, however, only if

(i) All the redundancy management mechanisms are designed correctly and operating reliably, and

(ii) In the implementation technology, the probability that more than M_N slices have failed in a redundant module M is less than the probability that the nonredundant counterpart of M has failed.

We have made these reliability assumptions in carrying out the investigation reported in this thesis.

1.2 Problem Statement

There are two major results reported in this thesis: a conceptual design for a fault-tolerant packet communication computer architecture, and a set of concepts and techniques for managing redundancy in self-timed hardware systems. The architecture design has been tailored specifically for data flow computer organizations and the fault tolerance features are built directly on program execution mechanisms at the machine level. The hardware modules in this architecture include *processing elements*, which provide storage for the program in execution and simple functional capabilities, *specialized functional units* for performing more complex operations, *routers* for constructing routing networks, and *allocators* for constructing allocation networks. The properties of

these modules are explained in more detail in Chapter 6. For now we note that the redundancy concepts and techniques we have developed for self-timed hardware systems are applicable to tolerating hardware failures in the class of packet communication systems described in [33], including all these hardware modules used in the data flow computing system. Thus while the redundancy management concepts and techniques have been developed to provide implementation tools for realizing the fault-tolerant architecture as specified, they are in fact more generally applicable.

Features of the data flow processor are described in Section 1.2.1. The need for introducing new concepts and techniques to manage redundancy in self-timed systems is illustrated in Section 1.2.2. Some of the technical problems that arise in implementing our redundancy management schemes in hardware are discussed in Section 1.2.3.

1.2.1 Design of a Fault-Tolerant Packet Communication Computer Architecture

Most computer systems [8], [30], [58], [67] designed to tolerate hardware failures are intended for high reliability or long life applications with modest computational requirements. In this thesis, we study fault tolerance techniques to cope with hardware failures in a multiprocessor designed to execute programs expressed in a subset of the data flow language presented in [22]. We shall refer to this multiprocessor system as a data flow processor for convenience. This data flow processor has several novel features:

1. *High performance and fault tolerance are achieved by using pools of identical hardware units.*

Fault-tolerant multiprocessor systems [11], [30] have been designed using multiple, identical, hardware units. A program running on the data flow processor is partitioned and stored on a set of identical processing elements. The data flow processor also has a homogeneous set of specialized functional units for performing complex operations. These functional units are allocated dynamically

to service requests from the processing elements. The dynamic allocation scheme provides direct support for graceful degradation with respect to these functional units. Programs prepared for execution on the fault-free data flow processor can run without modification if only a subset of functional units has failed.

2. *Communication among processing elements and functional units is supported by packet networks.*

In the data flow processor hardware modules serve two distinct functions - processing and communication. Processing elements in a data flow processor execute subcomputations concurrently. Communication between the processing elements is supported by packet networks, to be constructed out of a few basic LSI cell types. This architecture is quite different from most fault-tolerant computer architectures reported in the literature, which are bus-oriented von Neumann architectures [8, 59, 58] or bus-oriented multiprocessors [30, 67]. Store-and-forward packet network designs which can handle a large number of packets concurrently have been analyzed in [12]. In some of these networks the number of basic modules and the length of connections between them both exhibit faster than linear growth as the number of processing elements being serviced increases. It thus appears that a substantial amount of hardware in a practical data flow processor will be used to implement packet networks. The reliability of these networks will be an important factor in assessing system reliability and availability, and it is important to minimize the amount of redundant hardware invested in them to achieve a desired level of fault tolerance.

3. *Hardware in the data flow processor is organized as a packet communication architecture.*

The data flow processor hardware is organized by a *packet communication discipline* to support concurrency and modularity. Computer architectures are commonly implemented as synchronous digital systems in which events in all modules are synchronized with reference to a common timing signal. Many fundamental fault tolerance techniques have been developed in this context. In contrast, a packet communication computer architecture is implemented as an interconnection of self-timed modules whose activities are synchronized through localized signal exchange, in accordance with the adopted packet communication protocol. Fault tolerance techniques for asynchronous systems have been demonstrated previously in a fault-tolerant clock design [16] and are reported in a paper on *synchronization voting* by Davis and Wakerly [17]. We have generalized these techniques to show that byte slicing and coding techniques can be used to mask and detect failures in self-timed systems under more general fault assumptions than the commonly adopted stuck-at fault model.

In this thesis we present a fault-tolerant data flow processor design based on dynamic redundancy. In a dynamic redundancy scheme redundant hardware is incorporated to support on-line fault detection. Upon detecting a hardware failure, normal processing is suspended. The system is diagnosed and repaired (possibly degraded). Normal processing is then resumed and subcomputations contaminated by failures are reexecuted.

The dynamic redundancy scheme uses a combination of hardware-implemented fault detection and packet encoding techniques to mask packet network failures in the fault-tolerant data flow processor. It uses a retransmission strategy which is conceptually similar to error control techniques used in packet switched computer networks like the ARPANET. Differences in failure characteristics and performance requirements between these two types of networks have led to different

implementation strategies. We have also developed strategies for incorporating redundant hardware into a packet network to support rapid repair.

In some computer systems hardware failures are masked using module replication and majority voting. Another example of hardware-implemented fault masking is the use of error correcting codes in bit-sliced or byte-sliced memory systems. The hardware redundancy techniques presented in this thesis can be used to incorporate redundant hardware to either mask or detect failures on-line in a self-timed hardware system. It is thus possible to design a fault-tolerant packet communication computer architecture based on hardware-implemented fault masking. The dynamic redundancy scheme we have developed offers the potential of considerable hardware savings, especially in the packet networks. Hardware-implemented fault masking offers two advantages over a dynamic redundancy scheme: the computation in progress need not be held up until recovery is completed, since hardware failures are masked on-line; and no additional programming effort is required to insert checkpoints into application programs. The first capability is important in meeting stringent response time requirements, in process control applications, for example, but is not essential for numerical computation. As we shall see, the dynamic redundancy scheme we have developed performs recovery at the machine instruction level, and hence has no impact on applications programming.

When detailed logic designs and hardware failure rates are available for a hardware implementation, alternative schemes may be carefully evaluated to determine their cost-effectiveness.

1.2.2 Redundancy Management in Self-Timed Hardware Systems

In this thesis a data flow computing system is implemented in a physical machine organized by a packet communication architecture. Our dynamic redundancy scheme must be supported by hardware-implemented fault detection. In Section 1.1 we have introduced a byte-sliced hardware structure for redundant packet communication modules. We will use this structure to incorporate redundant hardware for fault detection. Given a specification of the non-redundant module and the extent of physical failures to be detected, we can pick an encoding scheme and then design the byte slices accordingly.

Let us consider how the notion of encoding the output words of byte-sliced modules is used to deal with byte slice failures in a synchronous hardware system under the stuck-at fault model. In a synchronous environment the byte slices in a module are synchronized with each other, and with fault handlers monitoring the outputs of the module, via a system clock. Under the stuck-at fault model the output lines of a failed module become stuck at either a logical 0 or a logical 1. As long as the clock signals are generated in accordance with their timing specifications and module failures obey the stuck-at fault model, each signal presented to a fault handler by a byte-sliced module will have stabilized at a signal value representing either a logical 0 or a logical 1, when the fault handler is activated. The outputs of a module can thus always be interpreted by a fault handler as a bit vector specified either by the bit vector function associated with the module or by the functional composition of this bit vector function with a fault function. Faults can then be detected or masked by designing the fault handler to operate on its input bit vectors based on the capabilities of the encoding scheme.

Consider next the application of byte slicing and encoding schemes to deal with failures in redundant packet communication systems. First of all, there is the problem of modeling hardware

failures in packet communication modules, and how to design decoders, under these models, to decode inputs from fault-free slices properly, in spite of interference from failed slices. This is the problem of fault modeling and will be discussed in the next subsection. In this subsection we explain the problems of keeping fault-free slices in a redundant module *properly synchronized* and *consistent* so that their outputs can always be grouped together properly for decoding. We call these latter two problems redundancy management problems.

The synchronization requirements can be illustrated by analyzing the operating principles behind the synchronous byte-sliced organization, commonly known as a triple modular redundancy scheme, shown in Fig. 1.2. In this organization, a redundant module is constructed using three identical slices, and the decoders are majority voters. Failures in any single slice are masked in majority voters in successor modules. Operation of the majority voters and processing modules are

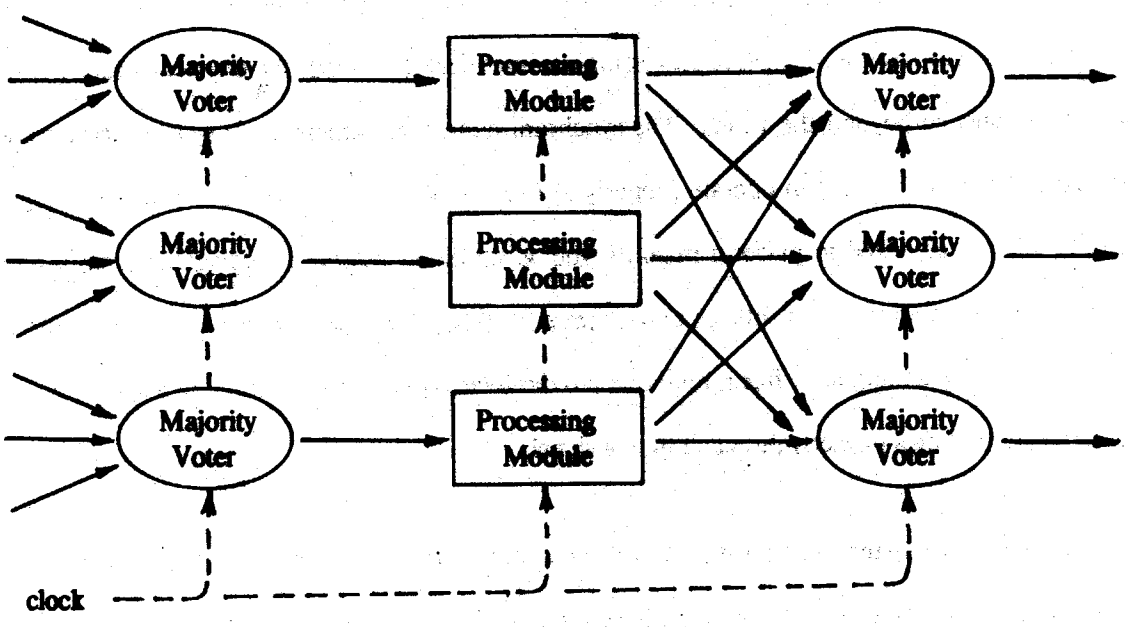


Fig. 1.2. A synchronous triple modular redundancy scheme.

synchronized through clock signals. Two implicit assumptions have been made in using the synchronous triple modular redundancy scheme to mask single module failures:

(1) The time skew in distributing a clock signal to the processing modules is bounded by a known constant.

(2) All fault-free processing modules in a triad will process identical inputs and generate the corresponding outputs in approximately the same amount of time.

When Assumption (1) is satisfied, processing modules in each triad will be activated to process a new batch of inputs from their input voters within a known time interval. Suppose the processing modules are activated to process new inputs by clock pulse C. Then under assumption (2), it is possible to calculate the time interval, in terms of the number of clock pulses, after which all fault-free processing modules activated at C will have presented their outputs to voters in successor modules. These voters are activated after this time interval to perform their fault-masking function. If either assumption is not met, it is possible for a voter to be activated by the clock signal before all input signals it receives from fault-free processing modules have stabilized. An erroneous datum may then be propagated beyond the voters. The synchronization requirement for a synchronous triple modular redundancy scheme to operate properly in masking single slice failures is that whenever a majority voter is activated to perform majority voting, the input signals it receives from fault-free processing modules must have stabilized to their proper values. This requirement defines the desired timing relationship among fault-free modules in a redundant system, and is assured in a synchronous system through using a properly designed timing signal.

In a self-timed system organized by a packet communication architecture, a majority voter in a triple modular redundancy scheme is activated upon receiving input packets from processing modules. During fault-free operation, a majority voter receiving input packets from processing

modules can be activated after receiving one packet from each processing module. There are hardware failures, however, which can cause a processing module to stop generating output packets. We call these failures *killing faults*. To detect and mask killing faults affecting a single processing module, a majority voter cannot always wait for an input packet from *each* processing module. After receiving input packets from two modules, it must decide at some point that the other module has failed, and start to generate alarm signals and construct the error-free data from the packets it has received. When it starts to decode the input signals, however, it must also be sure that packets from all fault-free processing modules have already arrived. Otherwise the same type of errors illustrated in Fig. 1.2 may occur.

We have developed an approach to manage redundancy in packet communication systems for fault tolerance through enforcing the following synchronization constraint:

> All input signals from fault-free modules feeding a decoder (the counterpart of majority voters in more general byte-sliced configurations) must have stabilized at their proper values whenever the decoder is activated to decode its input signals.

We refer to this requirement as the timing synchronization problem since it imposes certain timing constraints on fault-free byte slices and decoders receiving inputs from them. In our approach timing synchronization is maintained by assuring that fault-free byte slices feeding a decoder will always send outputs to the decoder within a fixed time interval of each other. This approach will be discussed in detail in Chapter 2.

At the end of Chapter 5, we describe briefly an alternative approach to organize redundancy in packet communication systems for fault tolerance and analyze its relative merits and disadvantages.

We have also studied another class of fault tolerance problems that arises in non-determinate packet communication systems. The notion of non-determinacy and the associated *consistency maintenance* problems are illustrated in the following scenario.

Consider a data base against which transactions from data terminals are processed. Suppose we now replicate the data base at several sites to mask single site failures, and it is desired to maintain consistency among them by serializing the transactions received at each site and ensuring that the streams of transactions fed to the data bases at these sites are all identical. An input transaction received at a terminal is forwarded to all the sites. Non-determinacy arises due to the variation in transmission delay along paths between terminals and data base sites, so that two transactions sent from different terminals may arrive at these sites in different order. To serialize concurrent transactions consistently even when some terminals have failed, data base sites must be able to communicate with each other. The fault tolerance problem we shall tackle is how to support such communication reliably, under the assumption that a failed data base site or failed terminal may exhibit such malicious behavior as sending inconsistent information to other sites.

Our approach to tolerate failures in non-determinate systems is based on a general algorithm for reliable message exchange discovered by Pease et al [45]. In our research the much simpler single-fault case was developed independently. The consistency problem is explained in Chapter 2, and its solution in Chapter 3 and Chapter 5.

1.2.3 Implementation Considerations

To implement fault-tolerant packet communication systems, we need a fault model which characterizes (i) the behavior of failed modules, and (ii) the interaction between hardware elements employed in the construction of fault handlers and outputs from failed modules. The latter part is necessary for evaluating the effectiveness of fault handlers, but is often ignored due to the fact that

the stuck-at fault model for logic gates is most commonly used. Under the stuck-at fault model, the output lines for faulty modules are assumed to be stuck at either logical 0 or logical 1, and it is straightforward to specify the interaction between fault-free logic gates and faulty input signals stuck at either 0 or 1. We will study hardware implementation of redundant packet communication systems using the stuck-at fault model, a *random pulse train* fault model and a *random wave train* fault model introduced in Chapter 4. The two latter models are chosen because they better capture the sensitivity of self-timed hardware modules to runt pulses and output hazards exhibited by their faulty neighbors. Their choice is also motivated by failure mechanisms in VLSI technologies which are not adequately modeled by classical stuck-at fault models [28], [63]. For both the random pulse train fault model and the random wave train fault model we have also specified the interaction between signals generated by failed modules and hardware elements used to implemented fault handlers so that the effectiveness of our fault tolerance techniques in combating hardware failures can be vigorously evaluated.

The problem of having metastable states in bistable devices, also known in the literature as the glitch problem or the synchronizer problem [14], [15], [37], [44] poses additional difficulties in implementing fault handlers under the random pulse train and the random wave train fault model. This problem often arises at the interface between a synchronous system and an asynchronous input. When a latch activated by an internally generated timing signal is used to receive an asynchronous input signal, the input signal may arrive at the latch almost simultaneously with the clock signal. The latch may then be driven into its metastable state. Once a latch has entered a metastable state, it can reside in that state for an arbitrarily long time. When the latch subsequently leaves this metastable state, its output may undergo a signal transition and violate other timing assumptions in the system. Metastable state phenomena have caused failures in synchronous digital systems. In a redundant packet communication system, a decoder must receive signals from every byte slice in a redundant

module. Under the random pulse train and the random wave train fault models, a failed byte slice may deliver faulty signals to drive receiver latches in neighboring decoders into their metastable states, causing these decoders to be out-of-sync by an arbitrary amount, with no finite bound.

In a self-timed system, another bistable device called an arbiter is often used to resolve conflicts in resource sharing. A two-input/one-output arbiter forwards input requests it has received at its output port. If two input requests are received simultaneously (within a short period of time), arbitration is performed and one of the requests will be forwarded. In performing arbitration an arbiter can also be driven into a metastable state.

To tolerate single arbiter failures, several of them must be operated together. If two conflicting requests are delivered to two failure-independent arbiters within a short interval of each other, these arbiters may enter and reside in their respective metastable states for different periods of time. There is thus no guarantee that they will both resolve the conflict within a fixed time period. This poses problems in assuring timing synchronization. The arbiter is also a source of nondeterminacy in packet communication systems. In the above scenario, there is no guarantee that the two arbiters will resolve the conflicts that arise in exactly the same way, and additional mechanisms are necessary to maintain consistency among them, so that the conflicting requests will be forwarded in the same order after arbitration has been performed independently by the two arbiters.

As we shall see in Chapter 5, metastable state problems, together with massive failure modes that affect several modules simultaneously, pose intrinsic limitations on the reliability that can be achieved when a self-timed system is enhanced for fault tolerance using the approach developed in this thesis.

1.3 Related Work

1.3.1 Fault-Tolerant Architectures

Redundant hardware for combating failures in electronic computers has been introduced in many different forms, ranging from quadded logic for protecting against single logic gate failures [61] to modular redundancy schemes for protecting against failures in entire processors, memories and buses. Review articles [53], [54] and [55] give accounts of the development of fault-tolerant computing in the past two decades. The pervasive use of computer controlled equipment, the high cost of maintenance and the lowered hardware costs brought about by VLSI have prompted much interest in improving system availability and maintainability through the use of redundant hardware. To date fault-tolerant computers systems are found mostly in telephone switching systems, space-borne vehicles, air traffic control and other applications where computer failures would incur high human or economic costs.

The fault-tolerant data flow processor design presented in this thesis uses a dynamic redundancy scheme to mask hardware failures: failures are detected on-line, the hardware system is repaired and the program in execution is restarted. Many computer systems have relied upon checkpoint/restart as a standard means of recovering from system failures. The STAR (Self-Testing and Repair) computer developed at Jet Propulsion Laboratory [8] is probably the first dynamically redundant fault-tolerant computer. In STAR each arithmetic unit, memory unit and data bus is protected by using error-detecting codes. Units performing logical operations are protected by duplication. When a failure is detected, the failed unit is replaced by a spare through power switching. Redundancy is managed by a configuration control unit, which is triplicated for fault tolerance. To mask hardware failures, checkpoints must be inserted into application programs from which program execution can be restarted after a failure has occurred. A set of operating system

procedures is provided specifically for checkpoint insertion. The STAR computer has been breadboarded and exercised through fault injection. It has evolved into the FTSC (Fault-Tolerant Spaceborne Computer) systems [52] currently under advanced development at Raytheon and the Aerospace Corporation.

A different approach to fault masking is adopted in the FTMP (Fault-Tolerant Multiprocessor) system developed at Draper Laboratory [30] and in the SIFT (Software-Implemented Fault Tolerance) system developed at Stanford Research Institute [67]. In both of these systems a program is executed on several hardware units and failures are masked by voting on the results of redundant computations. In the FTMP system, failures are masked through the formation of processor triads, memory triads and bus triads, and voting on the outputs of each triad with majority voters located in bus receivers. Spares are provided for replacing failed units. The SIFT system consists of a network of processing modules, each with its own local memory. In SIFT majority voting is performed by voting programs which receive results from redundant computations over the network. The number of processing modules dedicated to a task can be varied according to its criticality. In both FTMP and SIFT fault detection leads to system reconfiguration, but programs need not be rolled back to mask hardware failures.

Our work on hardware redundancy techniques has benefited from work on fault-tolerant clocking systems conducted under the FTMP project [16], and from work on consistency maintenance conducted under the SIFT project [45].

Many fault-tolerant computers in use have a bus-oriented internal architecture and are made fault-tolerant through providing redundant buses, redundant processing elements and redundant memory systems. The data flow processor contains a packet network which is absent in these architectures, and in general has rather different reliability and performance requirements.

Methodologies for incorporating fault tolerance into high performance computer systems and the associated technical problems have been discussed in [9], [10]. We have also found these discussions illuminating for our work.

The interconnection networks in a data flow processor are modular, one-way, packet routing networks. Each network serves a set of source modules and a set of destination modules (which need not be distinct). Packets are generated at source modules and delivered by the network to destination modules. Nodes in the networks are designed to forward packets only in the direction of packet flow, from sources to destinations. Other multiprocessor organizations, C.mmp [68] for example, use commutation networks for interprocessor communication. A commutation network is one which can be set up to satisfy a set of connectivity requirements between its input nodes and output nodes, but is otherwise passive. The fault tolerance properties of several classes of commutation networks have been studied in [36]. This work deals only with the topological aspects of fault-tolerant network design. For each class of commutation networks studied, it is shown how redundant paths may be incorporated so that faulty subnetworks can be bypassed. Several other redundant commutation network configurations are described in [42], as part of a fault-tolerant associative processor design. Neither of these works addresses the problems of hardware fault detection and coordinating network reconfiguration with recovery.

Packet routing using Banyan networks is proposed by Tripathi in [60]. He points out that a faulty node in a packet switched Banyan network can be bypassed if bidirectional paths are provided between network nodes, but gives no redundancy techniques or recovery strategies to support this scheme.

1.3.2 Synchronization and Consistency Maintenance

Our approach to maintaining timing synchronization is to assure that fault-free byte slices in the same redundant module will always deliver outputs to decoders receiving signals from them within a fixed time interval of each other. The techniques we have developed for this approach are closely related to techniques proposed for implementing fault-tolerant clocks. In a fault-tolerant clock several oscillators are operated in parallel to generate a set of periodic signals. These independently generated clock signals must, however, satisfy some in-phase requirements. One such requirement is that they must all change state within a fixed time interval of each other, just like the one we have imposed on byte slices in the same redundant module to maintain timing synchronization. A fault-tolerant clock design has been presented in [16]. In that design the circuitry is tailored to generating periodic signals. We will show how a fault-tolerant clock design can be derived from our synchronization techniques. Our synchronization techniques are also related to the *synchronization voting* technique given by Davies and Wakerly [17]. Davies [18], [19] has applied synchronization voting to fault-tolerant clock design, synchronization among redundant processing units, and synchronization among redundant microcomputers. We will also explain how timing synchronization can be achieved using synchronization voting under restricted fault assumptions.

Consistency maintenance problems in the fault-tolerant space shuttle computer have been discussed informally in a paper by Sheridan [56]. In this computer system the same process may run on several processor-memory units simultaneously. When an asynchronous interrupt occurs, the process may be interrupted at different points on different processor-memory units. Processes in the system communicate via shared memory. Suppose that an input interrupt occurs when a process is writing into a data area it shares with the input process, and that in one processor-memory unit it has finished writing into this area when it is interrupted while in another processor-memory unit it has not. The input process may then read different versions of the shared data in different

processor-memory units, and may then exhibit inconsistent behavior on these units. Sheridan describes a set of mechanisms for maintaining data consistency in this environment. Process incarnations running on different computers are required to synchronize with each other before accessing shared data. The fault model assumed and the effectiveness of this solution were unfortunately not discussed in any detail in [56].

In the SIFT system, inputs from different sensors, outcomes of diagnostic tests and internal clock readings are exchanged among processing modules on the network. It is assumed that a failed module may give inconsistent information to other modules. Pease et al [45] have developed a general algorithm for exchanging information among the processing modules so that

(i) the information transmitted by a fault-free module will be known to every fault-free module, and
(ii) the fault-free modules will agree on the contents of messages transmitted by faulty modules.

This algorithm requires at least $3f+1$ modules to tolerate up to f module failures among them. A negative result which states that this fault tolerance capability cannot be achieved with less than $3f+1$ modules is also proved in [45].

For our work we have independently developed the solution to the much simpler case in which at most one module may fail and each message contains the outcome of a binary decision. We have also studied the hardware implementation of this much simpler case in the self-timed systems environment.

1.4 Synopsis

In the following chapters, we first develop an approach to manage redundant hardware in a packet communication system for fault tolerance, and then present the design of a fault-tolerant data flow processor. Basic concepts and redundancy techniques are illustrated with hardware modules taken from the data flow processor design. Readers who wish to study the architecture of the fault-tolerant data flow processor before analyzing redundancy techniques for its implementation can also read the material covered in this thesis in that order.

Our approach to incorporate redundant hardware into packet communication systems for fault tolerance is introduced in Chapter 2. This approach is based on maintaining timing synchronization and consistency in redundant packet communication systems, even after hardware failures have occurred. The problems of timing synchronization and consistency maintenance are explained using redundant packet communication systems constructed with byte-sliced packet communication modules. Concepts and terminology for discussing these problems and their solutions more precisely are also presented in this chapter.

Robust algorithms for maintaining timing synchronization and consistency among byte slices in a redundant packet communication module are presented in Chapter 3. Under these algorithms, byte slices in the same redundant module send signals and messages to each other. A failed slice may send signals and messages of arbitrary contents at random. These algorithms are robust in that timing synchronization and consistency can be maintained in spite of hardware failures.

Hardware implementation of these redundancy management algorithms and decoder designs are studied in Chapters 4 and 5. In Chapter 4 we introduce a class of *asynchronous packet communication protocols* and three fault models. An asynchronous packet communication protocol defines the behavior of fault-free packet communication modules. A fault model characterizes the

behavior of failed modules as well as the interaction between signals generated by these failed modules and fault-free hardware elements used to construct fault handlers. In Chapter 5 we diacuss how to implement fault handlers, and other hardware modules, to support the fault tolerance strategy developed in Chapters 2 and 3. These hardware modules are designed to maintain timing synchronization and consistency, and to detect and/or mask hardware failures, under the fault models of Chapter 4.

A dynamic redundancy scheme for masking hardware failures in a multiprocessor architecture designed to execute parallel programs organized by data flow principles is presented in Chapter 6. Novel features of this computing system include its packet communication architecture, use of packet networks to support communication among processing elements and dynamic allocation of a homogeneous set of specialized functional units to service requests. Program organization and hardware module designs to support the dynamic redundancy scheme are explained. Strategies to incorporate additional modules and data paths into packet networks to support rapid repair are also described.

Application of the hardware redundancy concepts and techniques developed in Chapters 2 through 5 are illustrated by module designs given in Chapters 5 and 6.

A summary of results and concluding remarks are given in Chapter 7.

2. Timing Synchronization and Consistency Maintenance in Packet Communication Systems

The problems of timing synchronization and consistency maintenance have not received much attention in the literature of fault-tolerant computing. These problems are explained in detail in this chapter. Solutions to these problems, implementation techniques and hardware module designs are studied in subsequent chapters.

The concepts of byte slicing and output encoding have been introduced in Section 1.1. Timing synchronization and consistency maintenance problems will be illustrated with a byte-sliced organization for redundant packet communication modules. This organization is presented in Section 2.1.

We remind the reader that the basic problem in timing synchronization is to assure that whenever a decoder is activated to decode its input signals, all input signals from fault-free byte slices must have stabilized at their proper value. Our approach to maintain timing synchronization is to develop a set of mechanisms which can maintain *proper synchronization*. Informally, proper synchronization is achieved if bytes in the same input word, and acknowledgments for an input byte, are always delivered by fault-free slices in a redundant module to a neighboring decoder within a fixed time interval of each other. In Section 2.2 we introduce some concepts and terminology for characterizing proper synchronization among byte slices in a redundant packet communication module more precisely.

To use a byte-sliced organization and output encoding for fault tolerance, outputs generated by byte slices must also be *consistent*, in addition to being properly synchronized. Inconsistency among bytes in an output word generated by a byte-sliced module is due to non-determinate behavior. A

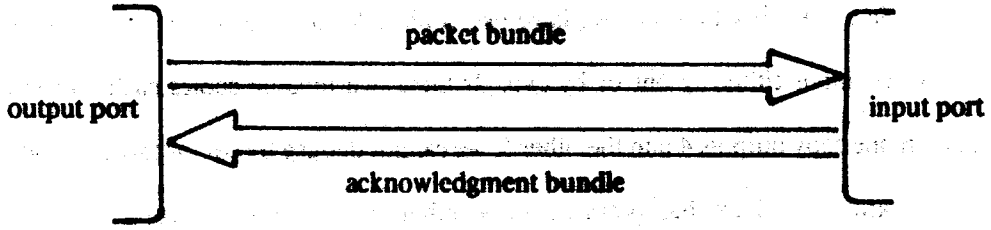
packet communication module is *determinate* if, starting from a given initial state, the set of output packet sequences it generates is a function of the set of input packet sequences it has received, independent of the order of input packet arrivals at its different input ports. A *non-determinate* system may generate different sets of output sequences after receiving the same set of input sequences, depending on the order in which packets arrive at its different ports, and on the outcomes of internal conflict resolutions. A determinate module has the nice property that its behavior is reproducible in both time and space. Two fault-free copies of a determinate module will generate identical output sequences when fed identical input sequences. Their outputs can hence be compared to detect failures among them. Transient failures can also be tolerated by feeding the same input sequence to a determinate module several times in succession, each time starting from the same initial state.

For now we illustrate how non-determinacy introduces inconsistency by another simple example. Suppose in a byte-sliced module we have two copies of a non-determinate module which increments an integer input either by one or by two. If these two module copies each receives the integer 3, and then one copy outputs 4 and the other 5, we cannot deduce by comparing their outputs alone whether a hardware failure has occurred or whether the two module copies have acted differently due to non-determinacy. Non-determinacy thus introduces additional complications in fault tolerance considerations. A primitive for introducing non-determinacy in packet communication systems and the associated consistency and fault tolerance problems are discussed in Section 2.3.

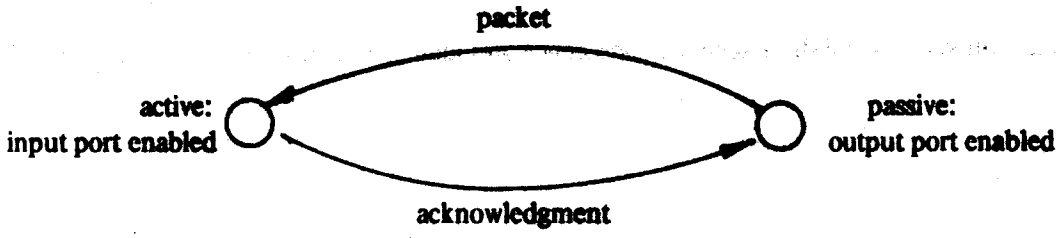
2.1 Byte-sliced Packet Communication Modules

A packet communication module interacts with its environment by transmitting and receiving packets at its *ports*. Each port consists of two sets of wires, carrying packets and acknowledgments into and out of the port, respectively. A packet communication *channel* (Fig. 2.1a) is constructed by connecting an output port of one module to an input port of another module.

Packet transmission over a channel can be described in terms of channel state transitions (Fig. 2.1b). Port activities cause a channel to alternate between an *active* state and a *passive* state. In an active state a packet is available over the channel from the sender output port. In this state the receiver input port is *enabled*. When the available packet is received and another packet can be transmitted over the channel, the receiver port resets the channel to its passive state. The sender port is then enabled and at some later time will change the channel state from passive to active and



(a) Hardware Structure



(b) State Transition

Fig. 2.1. Channel state transition.

transmit another packet over the channel. For the discussions in this chapter it is sufficient to note that a feedback mechanism is built into a communication protocol, so that during fault-free operation packets and acknowledgments alternate at each channel. Techniques for implementing asynchronous packet communication using signal transitions on wires in the packet bundle and the acknowledgment bundle will be presented in Chapter 4.

We next describe the hardware structure of a byte-sliced module and the structure of connections between two such modules. A slice (Fig. 2.2a) consists of a *processing module* (p-module), with each input port protected by a *synchronizing decoder* and each output port protected by a *fanout module*. We will refer to a decoder or a fanout module as a *control module*, to distinguish it from a processing module. Subsequent presentations are greatly simplified by grouping decoders and fanout modules in a redundant module together into *synchronization sets* (Fig. 2.2b), according to the input port or output port each protects. For such groupings to be meaningful, p-modules in a byte-sliced module must have the same number of input ports and output ports, and exhibit very similar packet processing behavior at these ports. The notion of schematic equivalence, as defined next, captures the desired relationship.[1]

Two packet communication modules are *schematically equivalent* if

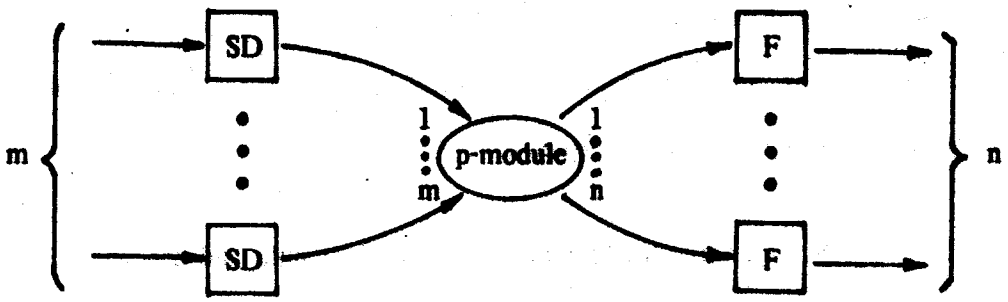
(1) They have the same number of input ports and output ports, and

(2) There exists a numbering scheme for input ports and output ports in each module such that if identical input sequences are fed to the i^{th} input port of both modules, for all i, then *the same number* of output packets will be generated at the j^{th} output port of both modules, for all j. (For

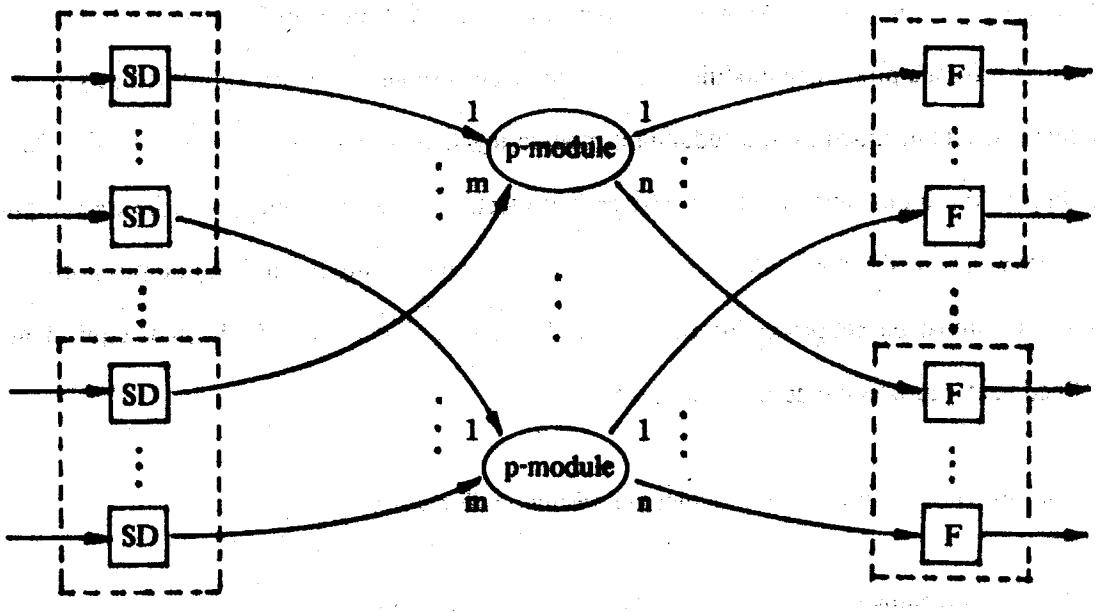
---

1. Note that the notion of schematic equivalence, and that of performance compatibility given later on, are well-defined only for fault-free modules.

SD -- Synchronizing Decoder F -- Fanout Module



(a) A byte slice.



(b) Synchronization sets in a byte-sliced redundant module.

Fig. 2.2. Structure of a byte-sliced redundant hardware module.

the hardware structures used in this thesis, the numbering scheme is always obvious from context and hence omitted.)

Copies of determinate modules are schematically equivalent and deliver identical output sequences when fed identical input sequences. Copies of non-determinate modules may not be schematically equivalent since the number of output packets generated at an output port of a non-determinate module may depend on the order of packet arrival at its different input ports, as well as on the contents of the packets it has received. Schematically equivalent modules need not be identical. It is straightforward to extend the notion of schematic equivalence to a group of more than two modules.

We will assume that byte slices in a redundant module are constructed out of schematically equivalent p-modules. A synchronization set (Fig. 2.2b) contains one control module from each slice. The common property among control modules in a synchronization set is that each protects the i^{th} input port of the p-module in its slice, or each protects the j^{th} output port of the p-module in its slice. Synchronization sets are enclosed in dotted lines in Fig. 2.2.

Connections between adjacent redundant modules are made by connecting a synchronization set of fanout modules in one to a synchronization set of decoders in the other (Fig. 2.3). Each synchronizing decoder receives input bytes from a set of fanout modules. Each input word to a synchronizing decoder consists of a number of bytes, one from each fanout module it is connected to. Note that each link shown in Fig 2.3 consists of a packet bundle carrying signals in the direction of the link, and an acknowledgment bundle carrying signals in the opposite direction. Every byte is transmitted as a packet between adjacent modules. Asynchronous protocols are also used by a p-module to receive packets from its synchronizing decoders and send packets to its fanout modules.

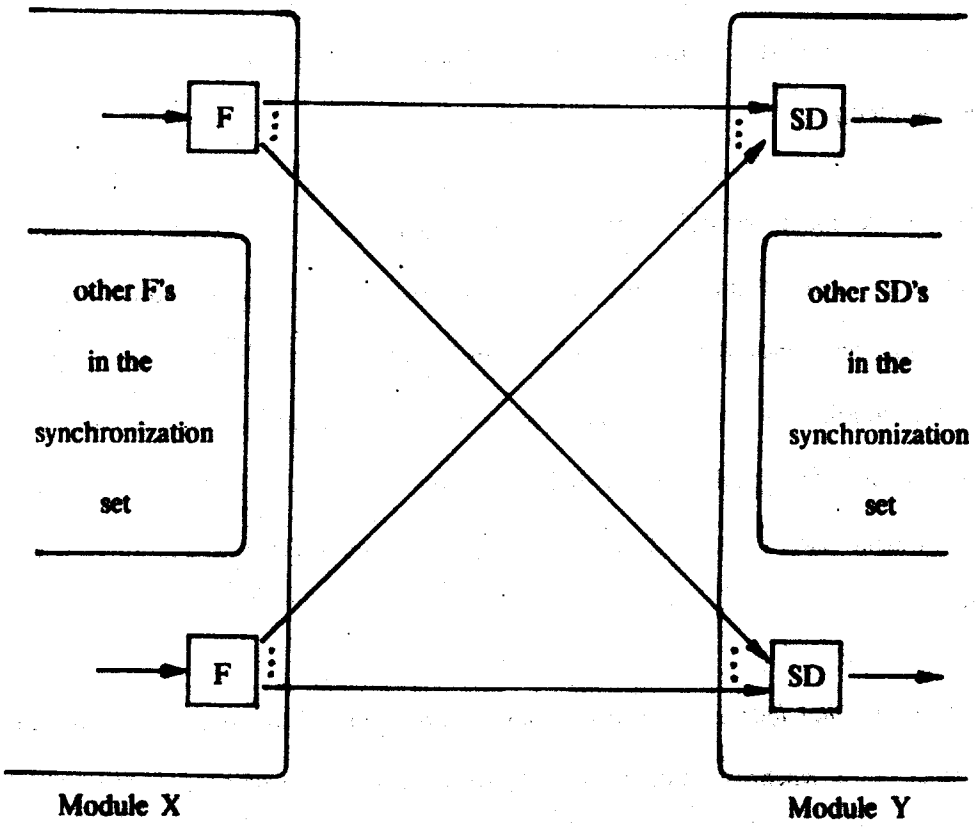


Fig. 2.3. Connection between synchronization sets in adjacent redundant modules.

A synchronizing decoder decodes input words received from a synchronization set of fanout modules. A fanout module forwards each packet it receives to the synchronization set of decoders it feeds. It also receives acknowledgments from these decoders and then returns acknowledgments to its p-module. P-modules perform packet processing. Let us examine the operation of byte slices in a redundant module in more detail.

Suppose each of the fanout modules in a synchronization set receives a byte from its p-module. These bytes together constitute an *output word* of the redundant module, and are distributed by the fanout modules to synchronizing decoders in a synchronization set in a redundant successor module.

Each input word received by a synchronizing decoder is thus delivered as a *batch of packets*, one from a different fanout module and each containing one byte of the input word. Failure of a single byte slice affects at most one byte of any input word received by a decoder. A synchronizing decoder decodes each input word received to derive an output word, which it then delivers to the p-module in its byte slice. After some time the decoder will receive an acknowledgment from its p-module. This acknowledgment is returned immediately by the decoder to fanout modules from which it has received input bytes. Each fanout module will thus receive a *batch of acknowledgments*, one from each byte slice which has received an input byte from it, and failure of a single slice will affect only one such acknowledgment. After a fanout module in the synchronization set has received acknowledgments from the successor module, it returns an acknowledgment to the p-module in its slice.

For subsequent discussions, we will also find it convenient to group packets and acknowledgments received and produced by p-modules in the same redundant module into *batches*. P-modules in a redundant module are schematically equivalent and will always be fed identical input packets. Packets and acknowledgments delivered to and generated by fault-free p-modules in a redundant module, under these conditions, can be partitioned into batches, according to the ports at which they are received or generated, and the positions they occupy in their respective streams. Packets, or acknowledgments, received at the corresponding ports of fault-free p-modules in a redundant module, and occupying the same position in their respective streams, belong to the same batch. Likewise output words, or acknowledgments, generated at the corresponding ports of fault-free p-modules in a redundant module, and occupying the same position in their respective streams, belong to the same batch.

2.2 Timing Synchronization

For a redundant packet communication system constructed out of byte-sliced packet communication modules, timing synchronization imposes two timing constraints on system operation:

(T1) Bytes in the same word generated by fault-free slices must have arrived when a decoder is activated to decode its input signals, and

(T2) Acknowledgments for the same output byte generated by fault-free slices must have arrived when a fanout module is activated to return an acknowledgment to its p-module.

Our approach to maintain timing synchronization is by maintaining *proper synchronization*:

A redundant packet communication system is properly synchronized if

(P1) Bytes in the same input word are always delivered by fault-free slices in a neighboring module to a synchronizing decoder within a fixed time interval of each other, and

(P2) Acknowledgments for the same output byte are always delivered by fault-free slices in a neighboring module to a fanout module within a fixed time interval of each other.

If proper synchronization is maintained, and there are enough fault-free byte slices remaining in each redundant module, then each decoder can determine whether it has received all bytes in the same input word generated by fault-free slices in a neighboring module. And after delivering a byte to synchronizing decoders in a neighboring module, each fanout module can determine whether all acknowledgments for that byte, generated by fault-free slices in the neighboring module, have been

received. The technique used in these control modules to achieve (T1) and (T2), given (P1) and (P2) will be presented in the next chapter.

Timing synchronization, as illustrated in Fig. 1.2, is achieved in synchronous systems by using byte slices which can process identical inputs in approximately the same amount of time, and by assuring that each clock signal is delivered to all byte slices in a redundant module within a fixed time interval. Our strategy for maintaining proper synchronization is also based on these ideas.

For a more precise discussion on timing synchronization, we next define *performance compatibility* for schematically equivalent modules, and *in-phase operation* for p-modules in redundant packet communication systems. The definition of performance compatibility is intended to capture the intuitive notion that two schematically equivalent modules are able to process identical inputs in roughly the same amount of time. In-phase operation is the approach we have adopted to ensure proper synchronization.

Performance Compatibility

Two p-modules M and M' are *performance compatible* if

(1) They are schematically equivalent, and

(2) If identical words, and acknowledgments, are always delivered to M and M' as a batch within a time interval bounded by ρ,[1] then packets and acknowledgments in the same output batch will be generated by M and M' within a time interval bounded by $\tau + \rho$, for some fixed τ.

---

1. We use lower case Greek letters to denote time intervals.

It is straightforward to generalize the definition of performance compatibility to a group of more than two schematically equivalent p-modules and to a group of control modules. We shall assume performance compatibility for determinate p-modules in a redundant module, and for control modules in a synchronization set. We will often say that two modules are *performance compatible within* τ, and call τ an upper bound on performance incompatibility.

Performance compatibility does not impose any constraint on how long a module may take to process an input, but only specifies the timing relationship among outputs delivered by schematically equivalent modules. We also note that two copies of a non-determinate module containing arbiters are not performance compatible since if either one of them enters a metastable state, it may reside in that state for an arbitrarily long time (Section 1.2.3).

In-Phase Operation

A packet communication system maintains *in-phase operation* of the byte slices in a redundant module if input words and acknowledgments in the same batch are always delivered to p-modules in these byte slices within a fixed time interval of each other.

For in-phase operation, we require that inputs in every batch delivered to p-modules in a redundant module be delivered within the *same* fixed time interval. We will also say that the outputs of a group of p-modules in a redundant module are in-phase if outputs in the same batch are always generated by the p-modules within a fixed time interval.

Our approach to maintaining proper synchronization is to maintain in-phase operation of p-modules in fault-free byte slices in every redundant module. A redundant packet communication system in which:

(i) byte slices in every redundant module are constructed using performance compatible p-modules,

(ii) in-phase operation of these byte slices is maintained,

will remain properly synchronized. We reason as follows. If in-phase operation of performance compatible byte slices in a redundant module is always maintained, then outputs of p-modules in fault-free slices must be in-phase. These outputs are forwarded through control modules in synchronization sets to neighboring modules. Since control modules in the same synchronization set are also performance compatible, bytes in the same word and acknowledgments for the same byte will be generated by these control modules, and hence by the fault-free slices, within a fixed time interval of each other. We make the further assumption that communication paths between synchronization sets have finite propagation delay. Proper synchronization (Properties P1 and P2) can then be maintained.

The synchronization techniques presented in Chapter 3 enable control modules in synchronization sets in a redundant module to maintain in-phase operation of byte slices in the redundant module locally. The strategy is to let control modules in the same synchronization set synchronize with each other before passing the inputs they have received on to the p-modules they are guarding. Using the synchronization techniques developed in Chapter 3, all fault-free control modules in the same synchronization set will deliver words or acknowledgments in the same batch to the p-modules within a fixed time interval, despite interference from failed modules.

2.3 Consistency Maintenance

Packet communication modules send streams of data packets to each other during system operation. A module is *determinate* if the set of output packet streams it generates is a function of the set of input packet streams it has received, independent of the temporal order in which input packets arrive at different input ports. A module is *non-determinate* if its output streams may depend on this

arrival order and on the outcomes of internal conflict resolutions, as well as on the data values carried in the input streams it has received. Non-determinacy introduces another dimension to the fault tolerance problem, namely, information replicated in redundant hardware must be kept consistent. The consistency problem is widely studied in distributed data base systems. Sheridan [56] discusses the problems of maintaining *data consistency* during instruction execution on the Space Shuttle control computer, with a program structure supporting prioritized multiprocessing and interrupt-driven input/output. Pease et al [45] have given algorithms for achieving *interactive consistency* in exchanging clock readings, sensor readings and diagnosis results among multiple computers. They have formulated the consistency problem as the "Albanian Generals Problem" :

Several divisions of the Albanian Army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messengers. After observing the enemy, they must decide upon a common plan of action, either to attack or retreat. However, some of the generals may be traitors trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that

(1) All loyal generals decide upon the same plan of action, and

(2) The loyal generals cannot be persuaded to follow the plan of action contrived of by a small number of traitors.

We have noted that the need for consistency maintenance is a consequence of non-determinacy. In the Albanian Generals problem, each loyal general has to independently decide whether to attack or retreat. Loyal generals can thus make conflicting decisions, leading to non-determinacy.

In this thesis we study consistency and fault tolerance for one specific form of non-determinacy in packet communication systems, introduced by combining two streams into an input stream to a

determinate module, using a *merge* module (Fig. 2.4). If two packets arrive at the two input ports of a merge module within a short time interval, the merge module may output them in either order, and may furthermore take an arbitrarily long time to decide which packet to output first. When input packets arrive further apart in time, the merge module outputs them as they are received.

In a redundant packet communication system, merging two input streams is implemented using a redundant module containing merge modules as p-modules in its byte slices. In this setting, identical input packets are always delivered to merge modules in the same redundant module within a fixed time interval, as a result of maintaining in-phase operation in the system. The consistency problem is to ensure that the output streams of non-faulty merge modules in a redundant module are identical. The fault tolerance problem is to ensure consistency and in-phase operation in the presence of hardware failures. The merge modules used in a redundant module also exchange messages with each other to achieve consistency. We shall call them *synchronizing merge* modules in the sequel.

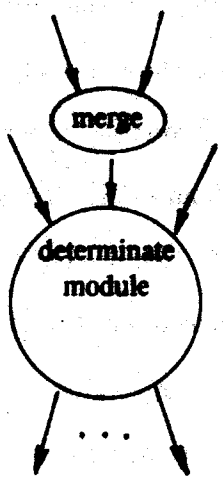


Fig. 2.4. Structure of a non-determinate module.

The strategy to achieve fault tolerance is to allow synchronizing merge modules to exchange messages with each other, informing each other of the input port from which a new packet has been received, and then reach an agreement with other synchronizing merge modules on which input port to use for forwarding the next packet. Note that it is not sufficient to receive a message from each synchronizing merge module in the redundant module and then perform majority voting on their contents to pick an input port. This is because the fault-free synchronizing merge modules can be split up just about evenly on which of two input ports to use, and faulty synchronizing modules can send different messages to different fault-free synchronizing modules, leading them to different majority vote decisions. We present an implementation of this strategy based on an algorithm for exchanging messages reliably among synchronizing merge modules in Chapter 3. This algorithm has been described by Pease et al [45] while we have developed the much simpler case for dealing with single synchronizing merge module failures independently.

2.4 Discussion

We have introduced two redundancy management problems for self-timed packet communication systems in this chapter: timing synchronization and consistency maintenance. These are redundancy management problems in the sense that these problems arise only when hardware functions are implemented using submodules which must be failure-independent, and hence operate independently, and yet their outputs must be correlated in some well-defined manner, such as by an encoding scheme. If hardware is immune from failures, hardware functions can be implemented in packet communication modules using techniques described in [4], [31], [33], [39] for which such problems do not arise. For applications requiring fault tolerance, we need solutions to these problems which are robust, so that timing synchronization and consistency are maintained in spite of hardware failures. By maintaining proper synchronization and consistency in a redundant packet communication system, a concrete foundation is provided for using byte slicing and encoding

techniques to detect and/or mask byte slice failures in self-timed hardware systems.

The problems of timing synchronization and consistency maintenance are illustrated with the byte-sliced module organization presented in Section 2.1 for redundant modules. Timing synchronization will be a problem in any fault-tolerant computer system constructed out of independently clocked processor-memory units. Consistency maintenance arises in sections of fault-tolerant computer systems that receive asynchronous inputs. These problems have not received much attention in the literature, perhaps due to their infrequent occurrence. It is nonetheless important to understand the nature of these problems, so that fault tolerance mechanisms for handling them are omitted only through judicious choice, and not from ignorance.

Timing synchronization is also a fundamental problem facing the designers of fault-tolerant clocks. In a fault-tolerant clock several oscillators are operated in parallel to generate a set of periodic signals. These independently generated clock signals must, however, satisfy some in-phase requirements. Synchronization can only be achieved by exchanging signals among oscillators, and must be maintained when a faulty oscillator stops oscillating or starts making signal transitions at random. The timing synchronization algorithm presented in Chapter 3 may be viewed as a generalization of the techniques described in [16] for implementing fault-tolerant clocks.

In a synchronous system intermodule activities are synchronized by referencing a common timing signal. After receiving an input each module is required to generate the corresponding output within a fixed time interval defined with respect to the timing signal. This implies performance compatibility among byte slices in a redundant module since they all must satisfy the same input/output timing constraint. In-phase operation is ensured as a consequence of the implicit assumption of negligible variations in propagation delay in distributing the timing signal from a common source to its numerous destinations. Fault handlers, by synchronizing with other modules

through this same timing signal, need only deal with data faults and code violations.

The input/output specification of a self-timed module imposes only sequencing and causality constraints on an implementation. For fault tolerance we have to reintroduce the notion of performance compatibility and to incorporate mechanisms to maintain in-phase operation in self-timed systems. Time metrics are thus reintroduced into an otherwise completely self-timed system. The additional complexity needed to deal with inconsistency is also a consequence of adopting the self-timed discipline. This is because arbitration is not needed in a synchronous system: conflicting requests can always be resolved through a predefined priority assignment.

Our whole approach to redundancy management for fault tolerance is based on maintaining timing synchronization and consistency in redundant packet communication systems. A methodological issue that arises is whether these two properties are necessary for tolerating hardware failures in packet communication systems. Its seems that consistency must always be maintained in a byte-sliced module, otherwise a packet batch generated by fault-free slices can have arbitrary contents, and byte slice failures can lead to undetected and uncorrected errors. The necessity for timing synchronization is further discussed at the end of Chapter 5, where we examine an alternative strategy and its relative merits and disadvantages.

3. Robust Algorithms for Timing Synchronization and Consistency Maintenance

In this chapter we introduce the basic algorithms used in our approach for maintaining timing synchronization and consistency in redundant packet communication systems. We maintain timing synchronization by maintaining in-phase operation of byte slices in a redundant system (Section 2.2). We maintain consistency by assuring that fault-free synchronizing merge modules in the same redundant module generate identical output streams (Section 2.3).

The algorithm for timing synchronization is implemented in every synchronization set for maintaining in-phase operation. Under this algorithm, control modules in the same synchronization set exchange signals with each other after receiving bytes in a new input word, or acknowledgments for a previously delivered byte, so that fault-free control modules in that set can deliver packets and acknowledgments in the same batch to p-modules within a fixed time interval of each other. The algorithm for consistency maintenance allows synchronization merge modules to exchange messages with each other to jointly determine the input port from which the next output packet should be taken. These algorithms must be robust in that in-phase operation and consistency must still be maintained in spite of malicious interference from failed modules.

For presenting these algorithms in this chapter, we use an abstract fault model under which a failed slice may generate signals and messages of arbitrary contents at random, or stop generating them altogether. Implementation of these algorithms, and circuit design techniques to assure that signals processed under these algorithms in control modules correspond to the abstract fault model used in this chapter, are studied in Chapter 5. The algorithms for maintaining timing synchronization and consistency are presented in Section 3.1 and 3.2, respectively.

3.1 An Algorithm for Timing Synchronization

Let us refine our design methodology given in Section 1.1 for constructing redundant packet communication systems. First of all, the non-redundant system is designed and specifications for each module in the non-redundant system are determined. Then an output encoding scheme is picked. To construct the redundant system, specifications for the p-modules in each redundant module are derived from the functional specifications of the non-redundant module and the chosen encoding scheme. P-modules, other than synchronizing merge modules, can then be implemented using techniques given in [4], [31], [33], [39]. Each p-module in a redundant module is extended into a byte slice by protecting its input ports and output ports with synchronizing decoders and fanout modules. These slices are then connected together by adding communication paths between control modules in the same synchronization set to form a redundant module.

In our approach to constructing redundant packet communication systems, we design each redundant module so that for each module we can specify:

(1) its functional specification,

(2) for each synchronization set in that module , an upper bound on the phase difference among bytes in the same output word, or acknowledgments for the same byte, generated by that synchronization set. This upper bound is by design an intrinsic property of the module, independent of any property of its neighbors.

We also design each control module so that timing synchronization (Properties T1 and T2 in Section 2.2) can be achieved once an upper bound for phase difference among packets and acknowledgments in the same input batch is known.

In constructing a redundant system, two redundant modules are connected together by connecting a synchronization set A in one to a synchronization set B in the other, as shown in Fig. 2.3. Given the specified upper bound on the phase difference among bytes in the same output word, or acknowledgments for the same input byte, generated by synchronization set A, and the variations in propagation delay along paths connecting A to B, we can calculate an upper bound on the phase difference among inputs in the same batch delivered to B. This derived upper bound is the same for all input batches sent from A to B. Likewise an upper bound is determined for batches of packets or acknowledgments sent from B to A. These upper bounds on phase differences are then used to "adjust" each control module to assure timing synchronization.

After a redundant system so constructed goes into operation, proper synchronization is maintained since the phase difference among output batches generated by a redundant module is always bounded by a constant which can be derived from parameters in the redundant module alone, independent of its interaction with its neighbors. As long as proper synchronization is maintained, timing synchronization is also maintained.

To support our design methodology, we need techniques for synchronizing byte slices so that the upper bounds specified in (2) above indeed exist, as well as techniques for designing control modules which can achieve timing synchronization given these upper bounds. In this section we develop an approach that can satisfy both of these requirements. In this approach, the basic cycle of activities carried out in a control module, regarding synchronization, consists of several steps:

For a synchronizing decoder,

(1) receive an input byte from each fault-free byte slice it is connected to, timing out faulty slices,

(2) synchronize with other decoders in its synchronization set,

(3) forward a word, deduced from the input bytes it has received, to the p-module it guards,

(4) receive an acknowledgment from the p-module and return the acknowledgment to all byte slices

it receives bytes from.

For a fanout module,

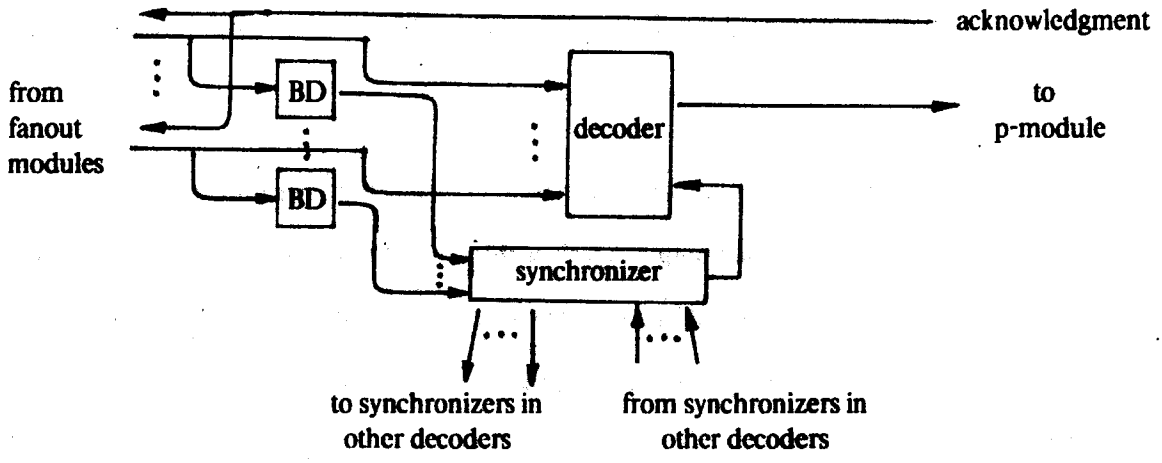
(1) receive a byte from the p-module in its slice, and forward the byte to all decoders in the synchronization set it feeds,

(2) receive an acknowledgment from each fault-free byte slice it is connected to, timing out faulty slices,

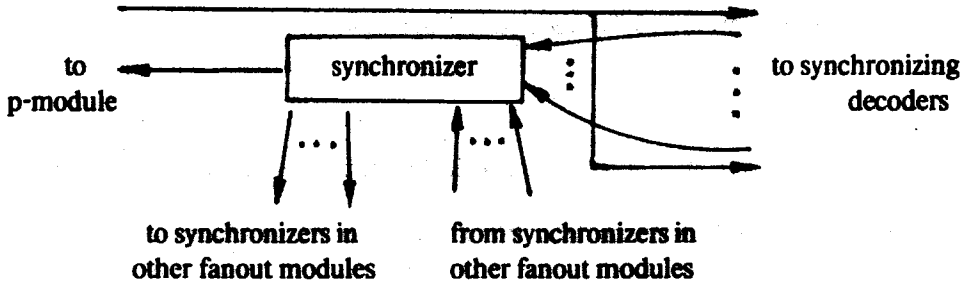
(3) synchronize with other fanout modules in its synchronization set,

(4) return the acknowledgment to the p-module.

The synchronization algorithm is implemented in *synchronizers* in control modules. A synchronizing decoder (Fig. 3.1a) consists of a synchronizer section and a decoder section. The synchronizer section implements the synchronization algorithm. The decoder section decodes the input word. The synchronizer section consists of a number of byte detectors (BDs) and a synchronizer. The byte detectors are designed to receive packets in the adopted communication protocol. Asynchronous packet communication protocols are discussed in Chapter 4.1. For this chapter it is sufficient to note that arrival of each new byte at a byte detector causes a signal to be sent on its output line to the synchronizer. Arrival of a new input word is signaled to the synchronizer as a batch of signals, one from each byte detector. By using byte detectors, the synchronizer can be designed independently of the packet protocols adopted. The synchronizer generates an output signal for each new batch of input signals it receives. In a synchronizing decoder, this signal activates the decoding section. Outputs of the decoder section are subsequently delivered to the p-module in its byte slice. Each acknowledgment received by the synchronizing decoder is returned to the fanout modules from which it receives input bytes.



(a) Synchronizing decoder



(b) Fanout module

Fig. 3.1. Hardware structure of control modules.

Operation of a fanout module (Fig. 3.1b) is similar to that of a synchronizing decoder, except that no input decoding is necessary, and acknowledgment signals returned by a neighboring module are received by its synchronizer directly. In a fanout module, the output signal of the synchronizer is returned as an acknowledgment signal directly to the p-module in its slice.

To support the requirements of our design methodology, it is sufficient for the synchronizers to guarantee that:

(S1) For each new batch of input signals received, all fault-free synchronizers in the same synchronization set will generate output signals within a fixed time interval of each other. The duration of this interval is bounded by a constant which can be calculated from the time it takes a synchronizer to perform basic operations, and each basic operation can be performed in a fixed time.

(S2) If input signals in the same batch are generated by fault-free neighboring slices within a fixed known time interval of each other, then for each new input batch, a synchronizer will generate its output signal only after it has received all signals in that batch generated by fault-free slices.

In this section we describe a synchronization algorithm for developing a synchronizer implementation with these properties. For this description, we assume that:

(A1) Each redundant module is constructed out of $3f+1$ byte slices, up to f of which may fail.

(A2) Input signals in the same batch, generated by fault-free neighboring slices, are delivered to a synchronizer within δ of each other.

We will also use an abstract fault model which states that a faulty byte slice can send signals, either directly or indirectly through byte detectors, to a synchronizer at random. In particular, a failed module may stop sending signals altogether.

Since each control module communicates with its neighbors via a handshake protocol, the interaction between a synchronizer and its neighboring fault-free slices obeys the constraint that none of these neighbors will deliver another signal to a synchronizer until after the synchronizer has

generated an output signal in response to the previous batch of signals it has received from them. Operation of a (fault-free) synchronizer under the synchronization algorithm, after it has just delivered an output signal in response to the previous batch of input signals, is as follows:

For each batch of input signals received, a synchronizer sends exactly one signal to every synchronizer, including itself, and generates exactly one output signal. A synchronizer sends signals to other synchronizers and to itself only after it has determined that an input signal from a fault-free neighboring slice has been received by some synchronizer. Due to the assumption that up to f neighboring byte slices and f synchronizers may fail together, a synchronizer cannot be certain of this until after it has received input signals from more than f neighboring slices, or synchronization signals from more than f synchronizers in its synchronization set. As soon as a synchronizer has received input signals from $f+1$ distinct neighboring slices or synchronization signals from $f+1$ distinct synchronizers, it starts sending synchronization signals to itself and to other synchronizers. Once a synchronizer has received signals from $2f+1$ synchronizers (itself included), it waits δ seconds and then delivers an output signal.

□

Suppose a fault-free synchronizer receives synchronization signals from $f+1$ distinct synchronizers at t and as a result generates synchronization signals to other synchronizers, we assume that all synchronizers will receive these signals by $t + \rho$, where ρ is an upper bound on propagation delay through the corresponding data paths.

Lemma 3.1 Under assumption (A1) and (A2):

(i) the output signals of the fault-free synchronizers are in-phase within a fixed time interval, namely, 2ρ, and

(ii) No non-faulty synchronizer will deliver its output signal before receiving a signal from every non-faulty neighboring slice.

Proof:

(i) Suppose a non-faulty synchronizer receives $2f+1$ synchronization signals at time t_1 and delivers its output signal at $t_1+\delta$. At t_1 it must have received signals from at least $f+1$ non-faulty synchronizers. At time $t_1+\rho$ every non-faulty synchronizer will have received at least $f+1$ signals from these non-faulty synchronizers, and will respond by sending signals to other synchronizers if it has not already done so. At $t_1+2\rho$ each non-faulty synchronizer will have received at least $2f+1$ signals from other synchronizers and at $t_1+2\rho+\delta$ each will have delivered its output signal.

(ii) When a non-faulty synchronizer has received synchronization signals from $f+1$ distinct synchronizers, at least one non-faulty synchronizer will have received input signals from $f+1$ neighboring slices, and at least one of these slices must be non-faulty. Thus under the assumption that signals in the same batch are delivered by fault-free neighboring slices within δ, when a non-faulty synchronizer delivers its output signal δ time units after it has received synchronization signals from $2f+1$ distinct synchronizers, all input signals from non-faulty neighboring slices must have arrived.

□

Note that Lemma 3.1 holds as a result of the actions of non-faulty synchronizers. The algorithm guarantees in-phase operation regardless of the faulty behavior exhibited by failed

synchronizers. Even though up to f faulty neighboring slices and f faulty synchronizers can send signals to synchronizers at random, they cannot trigger synchronization and output activities in fault-free synchronizers. And even if they stop sending signals, they cannot delay such activities indefinitely. To support in-phase operation in a packet communication system with cyclic paths, output signals from non-faulty synchronizers must be in-phase within a *fixed bounded* interval after synchronization. For the above strategy this phase difference is bounded by 2ρ. To satisfy (S1), however, a synchronizer must be implemented in such a way that the operation of generating synchronization signals after receiving such signals from $f+1$ distinct synchronizers takes a fixed amount of time, independent of the phase difference among inputs to the synchronizer. Implementation issues are discussed in detail in Chapter 5.

In the proof of Lemma 3.1 we have assumed that synchronizers act instantaneously. If synchronizers do not act instantaneously, variations in logic and path delays through them is bounded by some τ. Phase difference among their outputs may then increase by τ, again a time parameter which depends only on hardware parameters internal to the synchronizer set. The phase difference among synchronizer output signals in the same batch can still be bounded by a parameter applicable to all batches, and cannot grow without bound.

We can also develop some insight on why our timing synchronization algorithm requires $3f+1$ synchronizers to tolerate up to f failures among them. Assume we have a total of S synchronizers. A synchronizer must generate an output signal upon receiving signals from $(S-f)$ synchronizers (including itself). This is because up to f synchronizers may fail and stop sending signals altogether. When a synchronizer generates an output signal, the other synchronizers should start sending signals to each other. We observe that if a synchronizer has received signals from $(S-f)$ synchronizers, f of these synchronizers may be faulty and hence may not send signals to other synchronizers. Thus when a synchronizer generates an output signal, other fault-free synchronizers may have received signals

from only ((S−f) − f) synchronizers. Our strategy thus calls for a synchronizer to send signals to other synchronizers upon receiving signals from ((S−f) − f) synchronizers. A faulty synchronizer may send signals to other synchronizers at random. To prevent f faulty synchronizers conspiring together from controlling the behavior of fault-free synchronizers, we must have ((S−f) − f) > f, i.e., S > 3f.

A similar analysis will show that it is sufficient for a synchronizer to receive input signals from 2f+1 neighboring slices to tolerate up to f failures among these slices under the abstract fault model. We have chosen to assume that a synchronizer receives input signals from 3f+1 neighboring slices to simplify the presentation. The construction of redundant modules using less than 3f+1 slices is further discussed in Section 3.3.

3.2 An Algorithm for Consistency Maintenance

A scenario for studying consistency maintenance problems in redundant packet communication systems was given in Section 2.2. In this scenario non-determinacy is introduced using *synchronizing merge* modules as p-modules in byte slices in a redundant module. In such a non-determinate module all control modules in the same synchronization set will use the timing synchronization algorithm presented in Section 3.1 to maintain in-phase operation of the synchronizing merge modules. Input words and acknowledgments in the same batch will thus be delivered to synchronizing merge modules in the same redundant module within a fixed time interval.

When every synchronizing merge module has a packet pending for output from one of its input ports, which we shall call the module's *request source*, they exchange messages with each other and jointly pick an *input source*. Each synchronizing merge module then forwards a packet from the input source to its fanout module. The synchronizing merge modules can choose any input port proposed by a fault-free synchronizing merge module to be the next input source, since packets will

be delivered to every synchronizing merge module at that input port within a fixed time interval. During the message exchange, faulty synchronizing merge modules may lie. Non-faulty synchronizing merge modules must be able to pick the same input source, in spite of liars. As long as fault-free synchronizing merge modules agree on their input sources, their output streams will be identical.

Note that this problem is different from the timing synchronization problem, in that modules participating in the synchronization activities must reach agreement on the outcomes of two-way decisions, rather than on when events have occurred. Our approach to solving this fault tolerance problem is based on an algorithm for exchanging messages among synchronizing merge modules, under which:

(1) For every synchronizing merge module M, faulty or not, all fault-free synchronizing merge modules will agree on M's request source.
(2) The request source proposed by a fault-free synchronizing merge module will be known to all other fault-free synchronizing merge modules and used in their decisions to pick an input source.

(1) assures that if each fault-free synchronizing merge module applies the same algorithm to pick an input source, based on the set of request sources it has deduced for other synchronizing merge modules, then all fault-free synchronizing merge modules will pick the same input source. (2) assures that the chosen input source indeed has packets pending, so that faulty synchronizing merge modules cannot cause all fault-free merge modules to pick one input port as their next input source while these fault-free merge modules have only received packets at the other input port.

An algorithm for exchanging request sources among a set of synchronizing merge modules to achieve (1) and (2), using $3f+1$ synchronizing merge modules to tolerate up to f failures among them,

is given by Pease et al in [45]. In this algorithm messages are exchanged in rounds, and the number of rounds of exchange grows linearly as the number of failures to be tolerated. This contrasts with our timing synchronization algorithm in which only one round of signal exchange is ever necessary. Pease et al have also proved a negative result which says that fewer than 3f+1 synchronizing merge modules are not sufficient for tolerating f failures among them, if (1) and (2) are to be achieved by exchanging unauthenticated messages alone.

For this thesis we will only discuss the single synchronizing merge module failure (f = 1) case in detail. This greatly simplifies the presentation, especially the implementation details. It is also the most important case for the maintained environment envisioned for our physics simulation applications, in which the mean time to repair is relatively short compared to the mean time to failure for any redundant module.

We will thus assume that there are four synchronizing merge modules, one of which may be faulty. We start at the point at which each synchronizing merge module has received a packet from an input port, called its *request source*, and present the algorithm the synchronizing merge modules use to jointly pick the input port, called the *input source*, for transferring the next packet to the successor module. We assume that a failed synchronizing merge module may send messages to other synchronizing merge modules at random, and that the messages they send can specify either input port as its request source. The algorithm also calls for timing out other synchronizing merge modules to avoid waiting for messages from faulty modules indefinitely. To set up these time out mechanisms, we need to know the phase difference among inputs in the same batch to the synchronizing merge modules, and take metastable state phenomena into account. The details of implementing these mechanisms will be discussed in Chapter 5.

Each synchronizing merge module executes the following steps to determine the new input source. Request sources and input sources are denoted by either $\underline{0}$ or $\underline{1}$.

(i) Broadcast its request source to all synchronizing merge modules, including itself.

(ii) Receive messages identifying their request sources from other synchronizing merge modules. This step is timed-out and either $\underline{0}$ or $\underline{1}$ is assumed for a late module.

(iii) Relay messages received in Step (ii) to other synchronizing merge modules, including itself. These messages have the format:

> "I am module i ($1 \leq i \leq 4$). Synchronizing merge module j ($1 \leq j \leq 4$) told me that it has a request from input port k ($\underline{0}$ or $\underline{1}$)".

(iv) Receive messages sent by synchronizing merge modules at Step (iii). This step is also timed-out and arbitrary values are again assumed for request sources in missing messages. Each synchronizing merge module i will have received four messages regarding the request source of module j, one from each synchronizing merge module (including itself). The contents of these four messages are entered into a row of a message table (Sample message tables are given in Fig. 3.2). Each element in this table is the request source which the synchronizing merge module named in its column heading claims to have received from the synchronizing merge module named in its row heading. Module i picks the request source for module j by ignoring the (j, j)-th entry in the table and voting on the remaining entries in the row named by j.

(v) Having determined four request sources, one for each synchronizing merge module, the one in majority is taken as the input source for the next output packet. In case of a 2-2 tie, pick $\underline{0}$ over $\underline{1}$.

This algorithm is illustrated by the example given in Fig. 3.2. Instead of numbering the synchronizing merge modules, we label them A, B, C and D. The non-faulty synchronizing merge modules A, B and C receive messages from request sources 1, 1 and 0, respectively. Messages sent by D can have arbitrary values. The last column in the table kept by a module gives the request source determined by that module in Step (iv) for each synchronizing merge module. The extra box in the lower right hand corner of each table gives the input source deduced in Step (v) from the request sources. Note that the contents of messages sent by D in Step (iii) does not affect any decision made

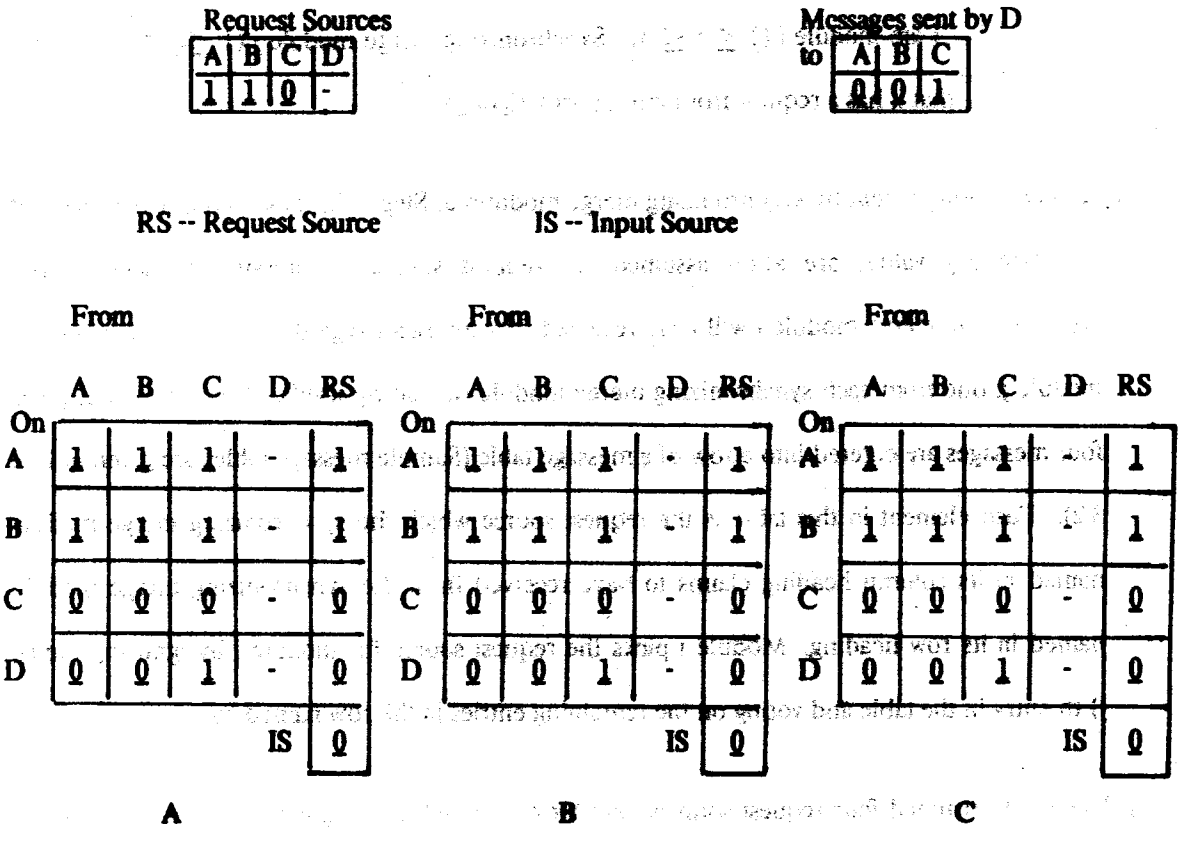


Fig. 3.2. An example to illustrate the decision algorithm.

by A, B and C in reaching agreement on an input source. The reader should also check that properties (1) and (2) stated at the beginning of this section are indeed satisfied.

The same tables shown in Fig. 3.2 will have been constructed if either one of the three modules A, B, or C times out module D in Step (ii) and assumes the corresponding value for D's request source. In this case, A, B and C would have picked the same input source in exactly the same way.

We next verify the algorithm informally. If a synchronizing merge module M is fault-free, all fault-free synchronizing merge modules (at least three, including M) will broadcast M's request source to each other, and hence will agree on M's request source among themselves. If M has failed, the information exchanged among the three other synchronizing merge modules regarding M's request source will be consistent. And again all non-faulty synchronizing merge modules will agree on M's request source. The reader is referred to [45] for the general algorithm and for a correctness proof.

3.3 Discussion

In the absence of a global timing reference, the timing synchronization algorithm presented in Section 3.1 allows control modules to maintain in-phase operation locally. This algorithm is closely related to the fault-tolerant clock design described in [16], and may be viewed as a generalization of the *synchronization voting* technique studied by Davies and Wakerly in [17]. These two references have been sources of ideas and examples for our work.

Fault-tolerant Clocking Systems

In the fault-tolerant clocking system described in [16] an array of identical oscillator modules is used to produce a number of phase-locked clock signals for global synchronization. Using 3f+1

modules, phase-locking is maintained on at least 2f+1 of the global clock signals after f failures. Local conditioning circuits at hardware units convert the 3f+1 global clock signals into local clock signals that are phase-locked if no more than f global clock signals have failed. Phase-locking among the global clock signals is maintained by conditioning each oscillator output to change state after either the elapse of a clock period or after sufficiently many other global clock signals have changed state.

Our synchronization algorithm presented in Section 3.1 can be used to implement a fault-tolerant clock in the configuration depicted in Fig. 3.3. In this scheme each synchronizer has a single input port and sends signals to other synchronizers as soon as it receives an input signal from this port. The output signal generated by the synchronizer is fed back to itself. The output signals generated by fault-free synchronizers can be out of phase by as much as 2ρ (Section 3.1). Each synchronizer, after receiving synchronization signals from 2f+1 distinct synchronizers (including itself), waits 2ρ before sending the next output signal.

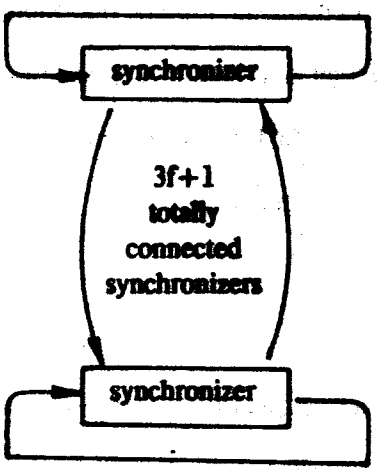


Fig. 3.3. A fault-tolerant clocking system.

Suppose each signal transition (either $0 \rightarrow 1$ or $1 \rightarrow 0$) on its output line represents an output signal generated by the synchronizer, then the configuration illustrated in Fig. 3.3 has the following properties:

As long as less than f of the synchronizers have failed,

(1) All fault-free synchronizers will generate output transitions in the same direction within 2ρ of each other.

(2) A fault-free synchronizer will make an output transition in one direction after all fault-free synchronizers have made their previous output transitions in the opposite direction.

These are exactly the phase requirements to be satisfied by redundant periodic signals generated by a fault-tolerant clocking system. Our synchronization algorithm thus suggests an approach to construct fault-tolerant clocks. Its practicality depends on whether or not it is possible to implement the desired properties of the synchronizer in a fault-tolerant fashion in the given hardware technology.

□

Synchronization Voting

Under the abstract fault model, a failed control module may send signals to only some synchronizers in its neighboring synchronization set, but not to others. For a restricted class of hardware failures, however, a failed control module will either not send any signals, or send one to each synchronizer within a fixed time interval. For such failures, timing synchronization can be achieved using the synchronization voting technique of Davies and Wakerly [17].

In synchronization voting, synchronizers in the same synchronization set are designed assuming that input signals in the same batch are in-phase within some δ. No communication is necessary among synchronizers. After receiving input signals from $f+1$ distinct byte slices, a synchronizer

waits δ seconds, and then generates an output signal.

For the restricted class of hardware failures under consideration, the phase difference δ among input signals in the same batch received by a synchronizer from fault-free neighboring slices depends only on:

(1) variations in propagation delay along paths connecting adjacent synchronization sets (Fig. 2.3),

(2) variations in propagation and gate delay among synchronizers in the same synchronization set,

(3) performance incompatibility parameters among p-modules in the same synchronization set.

δ can thus be computed for each synchronizer after the redundant system is designed and the above quantities are known, and used in its hardware implementation.

If a failed module can send signals to some synchronizers in its neighboring synchronization set, but not to others, then synchronization voting is not sufficient for timing synchronization. Suppose a set of synchronizers designed under synchronization voting receives a batch of signals from fault-free slices that are in-phase within δ. Under the more general fault assumption, one synchronizer may receive f+1 signals δ seconds before another. Output signals generated by these synchronizers can then be out of phase by δ. If the p-modules are performance compatible within τ, the next batch of outputs generated by fault-free byte slices containing these synchronizers can be out of phase by $\delta + \tau$. Under these conditions, phase difference among signal in the same batch can grow without bound along feedback paths.

The interested reader is referred to [17] for a more detailed discussion on synchronization voting.

□

Suppose an encoding scheme is picked for a redundant packet communication system under which 2f+1 byte slices is sufficient for detecting and/or masking failures in up to f slices among them. Timing synchronization can be maintained in such a system by using 2f+1 byte slices in each redundant module, and connecting the control modules in these 2f+1 byte slices as shown in Fig. 2.3. Each synchronization set in this configuration has 2f+1 control modules. We have already noted that the synchronizer algorithm can be designed to maintain timing synchronization using a total of 2f+1 input signals. To satisfy the requirement of having at least 3f+1 synchronizers participating in each round of synchronization activities, we can add f synchronizers to each synchronization set, connecting them to every control module in that synchronization set and to each other. These extra synchronizers do not receive input signals nor produce output signals. Their sole function is to enhance robustness in the synchronization algorithm, by relaying synchronization signals among all synchronizers. The algorithm they execute is:

Receive signals from f+1 distinct synchronizers, then send a signal to every synchronizer.

Finally, note that the steps in the algorithm for reaching agreement among synchronizing merge modules does not depend on the contents of the messages exchanged. The algorithm can also be used for reaching agreement among a group of modules making multiple-outcome decisions independently. In particular the consistency maintenance algorithm can be used to exchange clock readings among computer systems reliably, and hence for timing synchronization. This approach is clearly more expensive than our timing synchronization algorithm for managing redundancy in hardware systems, but it may be appropriate for synchronizing nodes in a computer network. The fault tolerance capabilities of this approach also depends on how hardware failures can be handled in an implementation.

4. Asynchronous Packet Communication Protocols and Fault Models

In the last two chapters we have explained our approach to constructing fault-tolerant packet communication systems. A redundant system is constructed by interconnecting redundant modules via redundant links. Each redundant module has a byte-sliced internal structure. Timing synchronization and consistency are always maintained, even in the presence of failures, in a redundant system. The output words of a byte-sliced module are encoded for detecting and/or masking failures in its byte slices. The emphasis in the last two chapters has been on explaining the problems of timing synchronization and consistency maintenance, the approach we have taken to sustain them, and the basic algorithms used in this approach.

In this and the next chapter we study hardware implementation issues. Since the effectiveness of our approach to fault tolerance in packet communication systems depends ultimately on our ability to implement the redundancy management algorithms given in Chapter 3, we will analyze implementation issues in detail. The basic concepts for control module design and analysis are presented in this chapter. Hardware implementation of control modules and synchronizing merge modules to support our particular approach to constructing fault-tolerant systems are studied in the next chapter.

A control module receives and generates packets and acknowledgments by interpreting and generating signal transitions on its input and output lines according to a set of conventions called an *asynchronous packet communication protocol*. Without knowledge of the adopted communication protocol, there is no basis for discriminating between normal and faulty behavior. We define a class of packet communication protocols in Section 4.1 whose use will be assumed in studying hardware implementation of control modules and synchronizing merge modules.

In Section 4.2 we present fault models for characterizing the behavior of faulty hardware modules. We describe the type of output signals that we assume a faulty module may generate on its output lines. The interaction between fault-free and failed modules is characterized formally by modeling the interaction between signals that may be generated by failed modules and hardware elements used to construct fault handlers. This latter aspect of fault modeling is essential for evaluating the effectiveness of fault handler designs under the assumed fault model, but is often left out under the stuck-at fault model.

For studying fault tolerance problems in packet communication systems, we feel that the stuck-at fault model is too restrictive, and have chosen to study hardware implementation issues using more general fault models which better reflect the sensitivity of self-timed hardware modules to runt pulses and output hazards. Hardware implementation of the control modules and synchronizing merge modules will be studied under the stuck-at fault model, the *random pulse train* fault model and the *random wave train* fault model. These latter two more general fault models are also motivated by failure mechanisms in VLSI technologies which are not adequately modeled by the stuck-at fault model. Interaction between faulty signals generated under these two fault models and hardware elements used to construct control modules and synchronizing merge modules are also specified.

4.1 Asynchronous Packet Communication Protocols

We have described packet transmission over a channel in terms of channel state transitions (Fig. 2.1). Both data and control information are sent over a channel. A *packet communication protocol* is a convention for interpreting the signal sequences transmitted over the wires of a channel to synchronize port activities. A packet communication protocol is *asynchronous* if the state of a channel can always be deduced by examining the signals carried on the wires in the channel, without consulting any external timing reference. When an asynchronous packet communication protocol is

adopted, a channel can reside in a state for an arbitrarily long period of time. In other words, an input port need not process an input packet within a fixed time interval after the packet is available and an output port need not generate output packets at a predetermined fixed rate. A packet communication module whose port activities are synchronized by asynchronous protocols is then, in this sense, a self-timed hardware module.

An acknowledgment bundle can be implemented using a single wire. Acknowledgments are delivered on this wire as signal transitions, as shown in Fig. 1.1b. For packet transmission on an n-wire packet bundle, each packet is represented by an n-bit binary string. Suppose a packet represented by string *a* is delivered to an input port, acknowledged, and then a packet represented by string *b* is transmitted. The packet bundle connected to the input port may undergo a series of intermediate state changes before settling down at *b*. The set of possible intermediate states that may occur when a packet bundle undergoes a state transition from *a* to *b* is characterized by the *subspace* covered by *a* and *b*, defined as follows:

Let $a = a_1...a_n$ and $b = b_1...b_n$ be two n-bit binary strings. The set of n-bit binary strings in the *subspace covered by a and b*, is defined as

$$\{ c = c_1...c_n \mid c_i = a_i = b_i \text{ for all i at which } a_i = b_i \}$$

The subspace covered by 00100 and 01110 is, for example,

$$\{ 0x_1 1 y_1 0 \mid x_1 \in \{0, 1\}, y_1 \in \{0, 1\} \}$$

To facilitate the design of input ports, we use an approach in which a stream of packets is transmitted on an n-wire packet bundle by encoding them alternately in two subsets, A and B, of the binary n-cube, and choose A and B so that transitions between them is easily recognizable. More specifically, A and B should be chosen so that an input port can recognize when a state transition on its input bundle, from any element in one set to any element in the other set, is completed. Our

approach is to choose A and B so that when a packet bundle undergoes a state transition from any element in one set to any element in the other set, no element in either A or B can ever occur as an intermediate bundle state. Any two sets of binary strings with this property are said to be *separable*, as defined next.

Consider an n-wire packet bundle. Let A and B be two subsets of the binary n-cube. A and B are *inseparable* if there exist three *distinct* strings a $\in$ A, b $\in$ B and c $\in$ A $\cup$ B such that c is in the subspace covered by a and b. A and B are separable if they are disjoint and not inseparable.

In other words, A and B are inseparable if there are elements a in A and b in B such that in a state transition on a packet bundle from a to b or from b to a, an element in A or an element in B may appear as an intermediate state. The two sets { 010, 111 } and { 100, 001 } are thus separable while { 000, 010, 111 } and { 100, 001 } are not. When two separable sets are used for packet transmission the receiving input port can detect the completion of the next packet transmission, after a packet encoded in either A or B is acknowledged, by looking for a string in the other set on the packet bundle.

For an asynchronous protocol based on two separable sets A and B, a packet bundle state is *illegal* if it is not in the subspace covered by A and B.[1] It is *unstable* if it is legal but is in neither A nor B. Elements of A and B are called *stable* states.

The ready/acknowledge handshake protocol is an example of an asynchronous packet communication protocol. The two separable sets are the singletons { 0 } and { 1 }, used on the *ready* wire (Fig. 1.1a) to control packet transmission. Packet contents are delivered on the data wires.

---

1. The subspace covered by two sets A and B is:
   $\cup$ { subspace covered by a and b | a $\in$ A, b $\in$ B }

Another example of an asynchronous packet protocol is the dual-rail protocol. In a dual-rail protocol every information bit in a packet is transmitted over a pair of wires. A packet bundle thus consists of n wire pairs. It is in a *spacer* state when all wires carry the logic value 0. It is in a *data* state when exactly one wire in each pair carries the logic value 1. The singleton set, consisting of only the spacer, and the set of all data states provide two separable sets for asynchronous packet transmission. A packet is transmitted over the bundle in a spacer → data transition. A data → spacer transition resets the bundle state for the next transmission. Under the dual-rail protocol, a packet bundle state is *illegal* if some wire pair carries 1 on both wires. Data states and the spacer state are *stable* legal states. Other legal states, in which some wire pair carries "00", are *unstable*. Channel state transitions for sending 1-bit packets in dual-rail is illustrated in Fig. 4.1. The two separable sets used in a 1-bit dual-rail protocol are { 00 }, and { 01, 10 }.

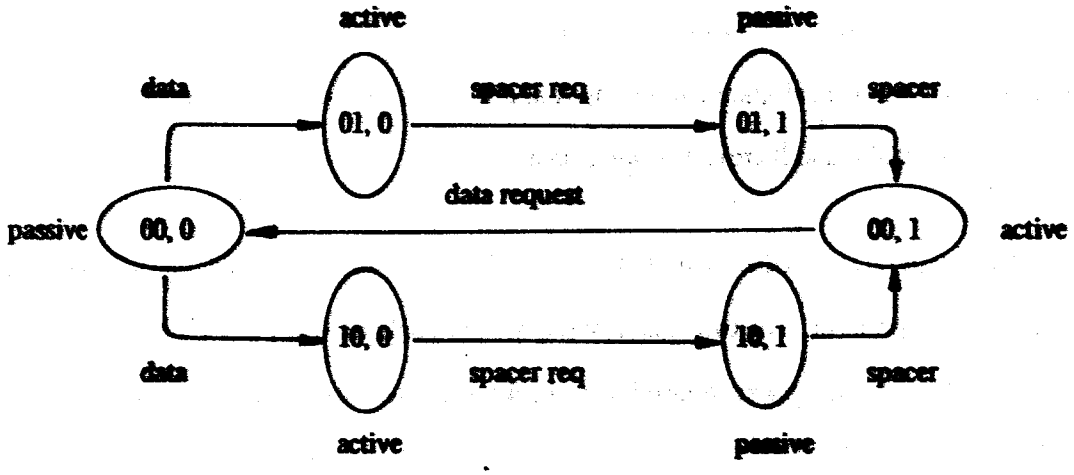


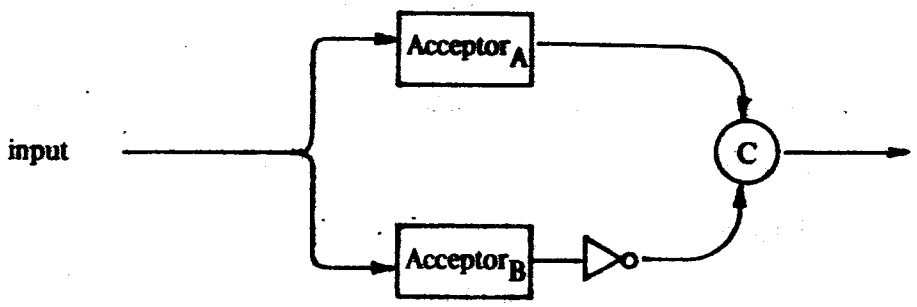
Fig. 4.1. The dual-rail protocol.

Other examples of asynchronous packet communication protocols can be found in [4].

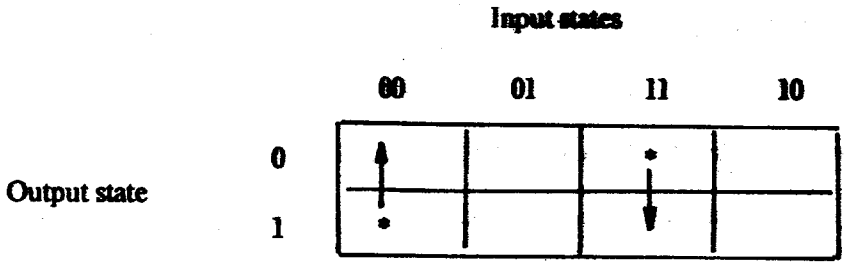
An input port can detect the arrival of a new packet using a packet detector (Fig. 4.2a). A detector is constructed out of acceptors for two separable sets A and B, and a Muller C-element. The output of the C-element keeps its present value as long as its two input values disagree, and changes to the common input value otherwise. Operation of the C-element is described by a transition diagram in Fig. 4.2 (b). A total state of the C-element specifies the signal values carried on its input and output wires. The unstable states of a C-element are marked by an asterisk in the transition diagram. A C-element acts by making transitions from unstable to stable states. An acceptor for a set S is a hazard-free combinational circuit whose output is 1 if and only if its input is in S. The output of a packet detector undergoes $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions as the monitored bundle makes transitions between the two separable sets. A packet detector for dual-rail protocols is shown in Fig. 4.2 (c). Packet detectors in synchronizing decoders are called byte detectors because each packet received by these decoders contains exactly one byte of each input word.

Further restrictions, based on the notion of Hamming distance [46], [65], for example, may be imposed on separable sets used in packet communication to support error checking and error correction. We next illustrate a technique that can be applied to translate familiar error detecting or error-correcting codes [46], [65] into a dual-rail code with the following example.

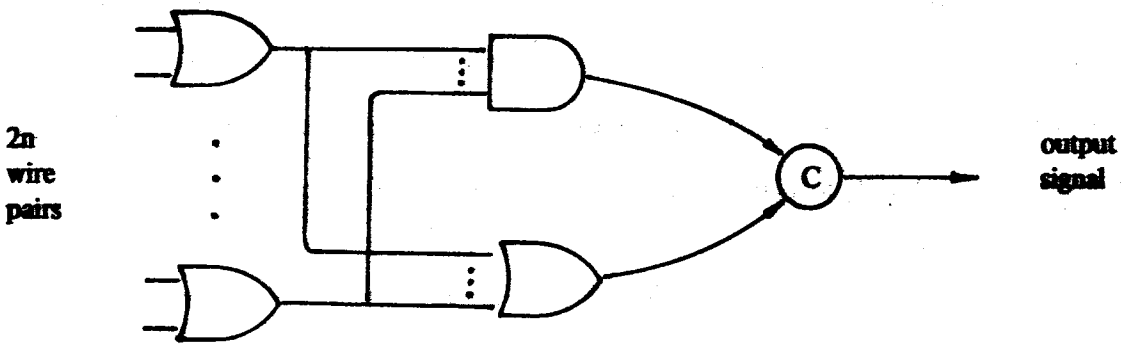
An n-bit dual-rail parity code partitions the data states of a packet bundle according to their parity. Let the dual-rail encoding of the information bits 0 and 1 be "01" and "10", respectively. A data state in a dual-rail protocol has even parity if it contains an even number of "10"'s, and odd parity otherwise. An even-parity dual-rail asynchronous packet communication protocol is just a dual-rail protocol whose data states all have even parity. In using a dual-rail even parity code, even parity packets alternate with spacers on packet bundles.



(a) Hardware structure of a packet detector.



(b) Transition diagram for the C-element.



(c) A detector for dual-rail protocols.

Fig. 4.2. Packet detectors.

If a dual-rail parity code is used to detect failures in a redundant module, each dual-rail bit should be generated by an independent slice to assure that common failure modes, such as single slice failures, affect at most one dual-rail bit. Encoding and decoding packets for fault tolerance are further discussed in Section 5.2.

4.2 Fault Modeling

We are interested in designing control modules and synchronizing merge modules that can tolerate hardware failures in their neighbors in maintaining synchronization and consistency among byte slices in a redundant module, and detect and mask data errors. To study these designs vigorously, we need to explicitly specify the interaction between signals generated by failed modules and hardware elements such as combinational gates, C-elements and latches, used in constructing fault handlers. We call such a specification a fault model. In using a given fault model in fault handler design, we are assuming that under the most common hardware failure modes the underlying physical interaction between signals generated by failed modules and hardware elements in fault handlers can indeed be characterized by the adopted fault model.

In this section we introduce the stuck-at fault model, the random pulse train fault model and the random wave train fault model. These latter two models are generalizations of the widely used stuck-at fault model. Several failure mechanisms in VLSI NMOS and CMOS technologies that cannot be modeled by classical stuck-at fault models are reported in [28], [63]. Some of these mechanisms are aging processes which modify the electrical characteristics of basic transistor circuits. It seems that as feature sizes, path widths and separation between active circuits reach submicron levels, the stuck-at fault model will not be adequate for modeling many on-chip failure mechanisms, or transient failures caused by external interference.

We note that the stuck-at fault model, the random pulse train fault model and the random wave train fault model form a strict hierarchy in terms of generality and modeling power, the random wave train model being the most general, the stuck-at model the most restrictive. In the next chapter we study the design of control modules and synchronizing merge modules using these fault models. For a specific implementation technology, a fault model can be validated either through experimental measurements or physics modeling. Validation of these fault models for available implementation technologies is beyond the scope of this thesis.

The Stuck-At Fault Model

Under this fault model a signal generated by a failed module appears to a hardware element in a fault-free module as if the signal is stuck at either the logic level 0 or the logic level 1. The behavior of a hardware element receiving a faulty signal is identical to that of the element under fault-free operation, with its corresponding input tied to either 0 or 1. This is the most common fault model assumed in studying fault-tolerant digital systems, and is often used to characterize logic gate failures caused by output lines shorting to ground or V_{CC}. This fault model is used in many production environments, semiconductor manufacturers, for example, for test generation.

A byte slice in a redundant module communicates with its neighbors via an asynchronous packet communication protocol. Our stuck-at fault model for byte slices states that a failed byte-slice behaves exactly like the fault-free slice except that some of its output lines are either ORed with V_{CC} or ANDed with ground (Fig. 4.3). Thus while some output lines of a failed byte slice are stuck, signals may still be generated on its other output lines in accordance with the adopted communication protocol. We will also explicitly assume that an output line of a failed byte slice can become stuck at a logical value only when the output signal on that line is already residing at that value, and thus no spurious signal transition is ever generated. Under the stuck-at fault model in Fig.

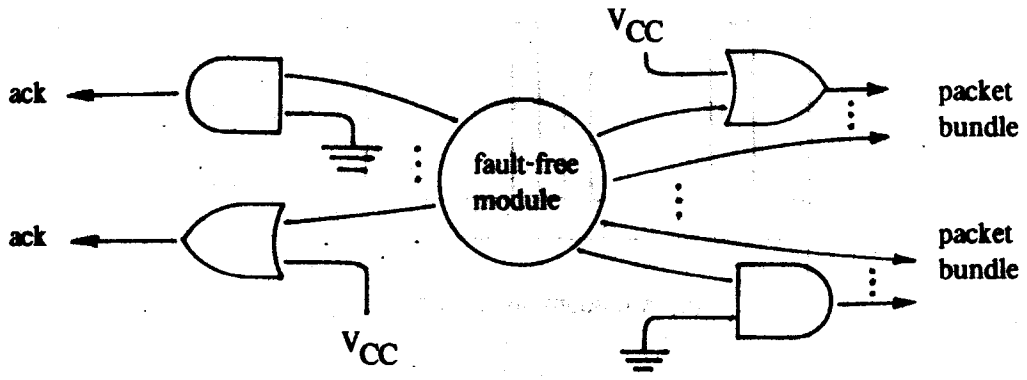
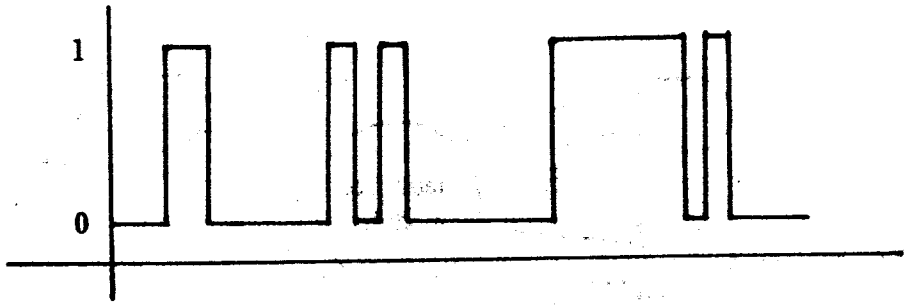


Fig. 4.3. The stuck-at fault model for byte slice failures.

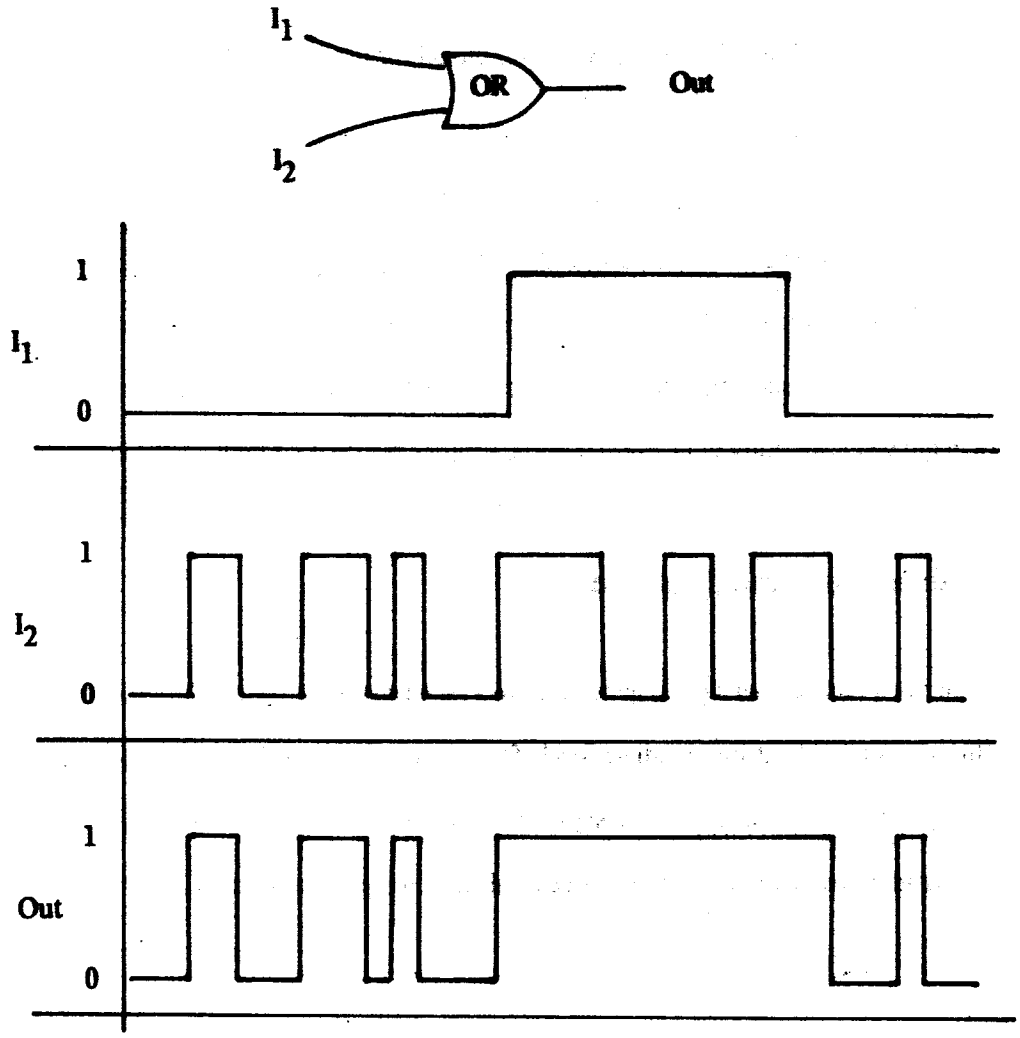
4.3, an output byte or acknowledgment generated by the failed slice can be delivered to some, but not necessary all, neighboring control modules. An acknowledgment wire stuck at either 0 or 1 will stop returning acknowledgments. But if some wires in a packet bundle (Fig. 2.1) are stuck, some packets generated by the failed slice may still be delivered on that bundle. These situations will be discussed in more detail when we study fault handler designs under the stuck-at fault model in Section 5.1.2.

The Random Pulse Train Fault Model

Under this fault model signals generated by a failed module may oscillate randomly between 0 and 1, but do not reside at intermediate levels for any significant period of time (compared to logic gate delays). We model such a faulty signal as a random pulse train (Fig. 4.4a). The interaction between a random pulse train and a fault-free hardware element is as follows:

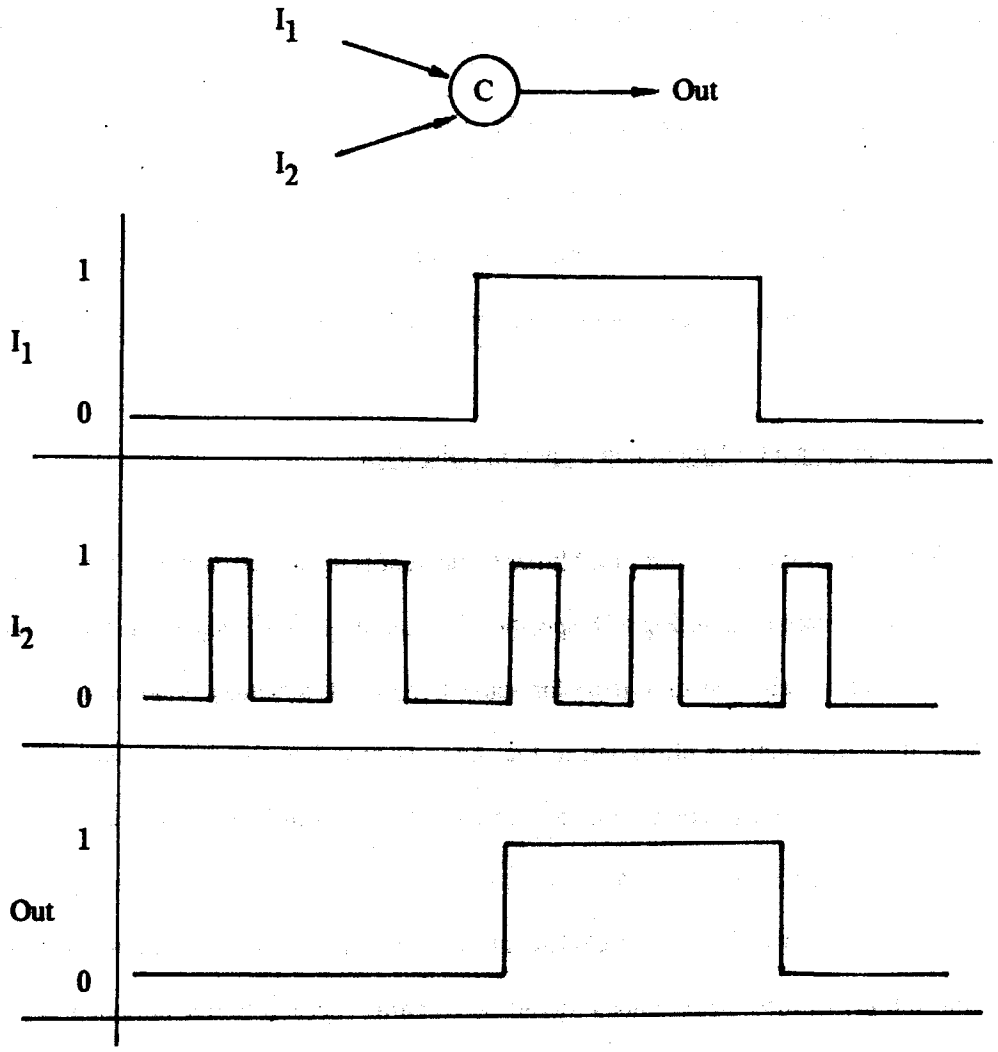


(a) A random pulse train.



(b) Interaction between a random pulse train and an OR gate.

Fig. 4.4. The random pulse train fault model.



(c) Interaction between a random pulse train and a C-element.

Fig. 4.4. The random pulse train fault model (continued).

Interaction between a combinational gate and a random pulse train

A combinational gate, such as an AND gate or an OR gate, is modeled as a logic gate whose output at time t is obtained by applying the corresponding boolean function to its input value at time t, followed by a delay element whose input signal is delayed by some given δ. Interaction between a combinational gate and a random pulse train input is illustrated using an OR gate in Fig. 4.4b.

Interaction between a C-element and a random pulse train

The input/output behavior or a Muller C-element is illustrated in Fig. 4.2. More precisely, we assume that the state table in Fig 4.2b specifies the output of the C-element at time t given its input and output signal values at time $(t-\delta)$ for some fixed δ. Interaction between a C-element and a random pulse train input is illustrated in Fig. 4.4c. In modeling this interaction we have made the assumption that signal transitions in a random pulse train are sufficiently far apart that a C-element, when activated, will always settle down in its new state before the next transition occurs, and that a random pulse train does not drive a C-element into a metastable state. Again, we caution the reader that the validity of these assumptions should be carefully checked before adopting this fault model for specific hardware implementation. Our key application for C-elements in control modules design under this fault model is for filtering out random pulses using a fault-free input signal, as illustrated in Fig. 4.4c.

Interaction between a latch and a random pulse train

For a latch receiving a random pulse train under the control of a fault-free latching pulse, we assume that the output of the latch will settle down at either 0 or 1 within a fixed time interval after the latching pulse. It is well known that if the input to a latch changes its value at about the same

time the latching pulse arrives, the latch may be driven into a metastable state and remain in that state for an arbitrarily long time. For many technologies the probability that a latch, after entering its metastable state at time t, remains in that state at $t+\delta$ decreases rapidly with δ. It is thus possible to improve the accuracy of the random pulse train fault model by assuming a longer settling time between delivering a latching pulse and reading the output of the latch. The probability that the latch will remain in its metastable state after the assumed settling time is, however, nonzero and must be taken into account in calculating reliability measures.

The Random Wave Train Fault Model

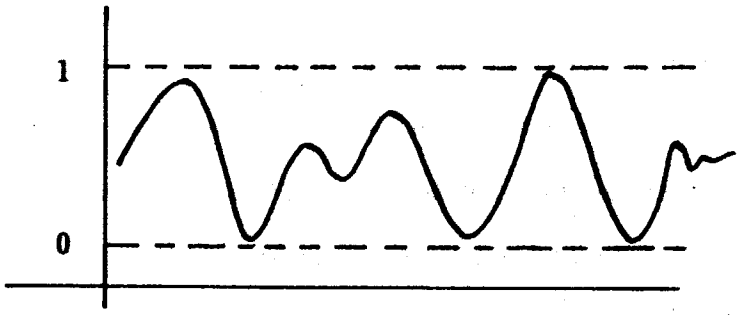
Under this fault model signals generated by a failed module can wander arbitrarily in the region bounded by the signal values 0 and 1. We model such a faulty signal as a random wave train (Fig. 4.5a). The interaction between a random wave train and a fault-free hardware element is as follows:

Interaction between a combinational gate and a random wave train

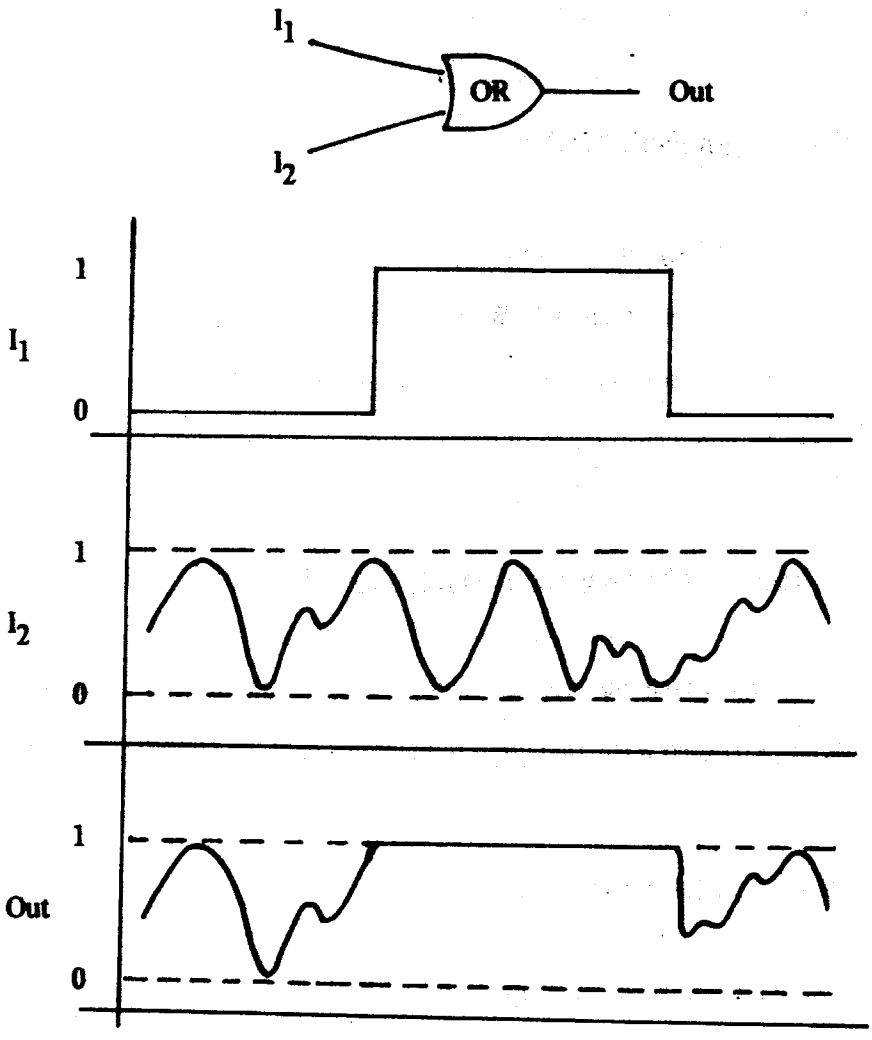
Same as that between a combinational gate and a random pulse train. Interaction between a combinational gate and a random wave train input is illustrated using an OR gate in Fig. 4.5b.

Interaction between a C-element and a random wave train

We assume that a random wave train input can be propagated to the output of the C-element, as illustrated in Fig. 4.5c. Thus under the random wave train fault model, C-elements are no longer useful for filtering faulty signals.

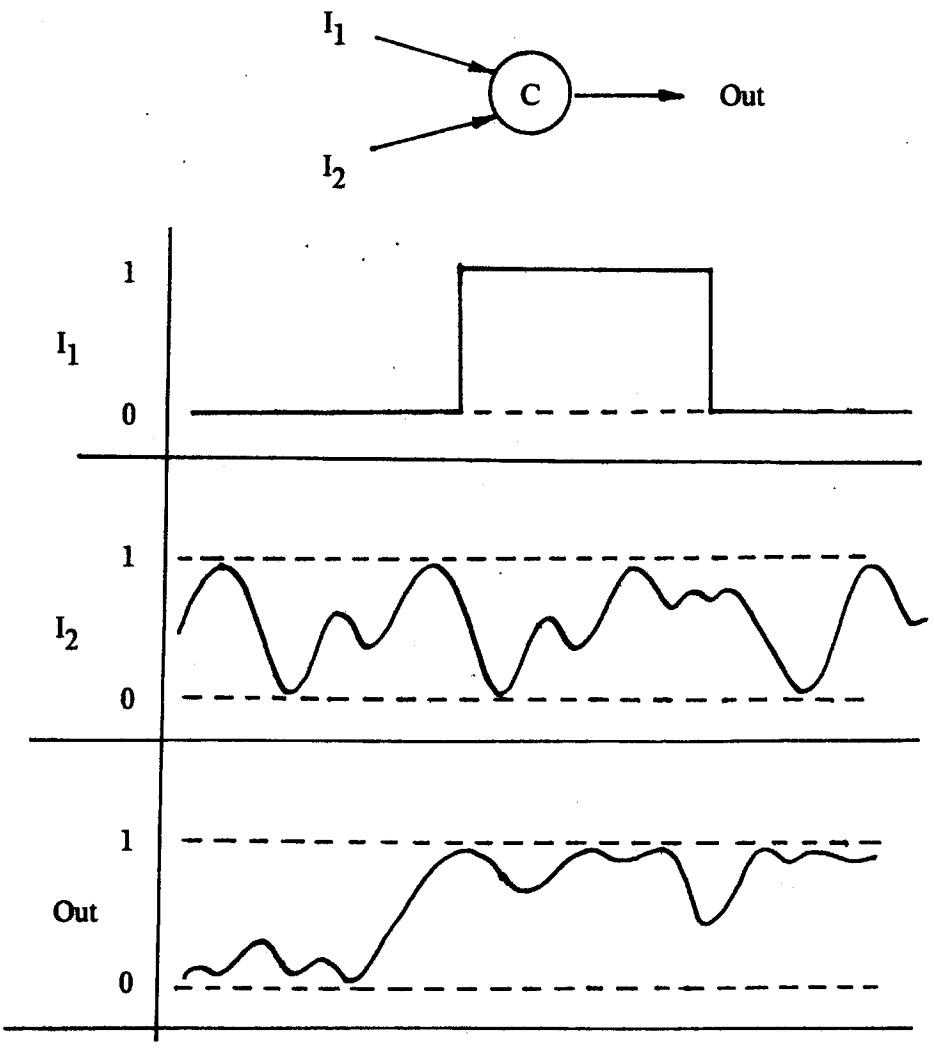


(a) A random wave train.



(b) Interaction between a random wave train and an OR gate.

Fig. 4.5. The random wave train fault model.



(c) Interaction between a random wave train and a C-element.

Fig. 4.5. The random wave train fault model (continued).

Interaction between a latch and a random wave train

Same as that between a latch and a random pulse train.

4.3 Discussion

A packet communication module interacts with its neighbors at several different levels. At the wire level it delivers signal transitions on wires to its neighbors. At the packet communication level it delivers packets to its neighbors over a bundle of wires and delivers acknowledgments over acknowledgment wires, in accordance with the adopted packet communication protocol. Higher levels of interaction, interpreting a packet as a request for service, for example, need not concern us here. The three fault models we have given specify the behavior of a failed module at the wire level. In the next chapter we will relate these faults on wire bundles to packet communication errors and discuss how these faults relate to the abstract fault models assumed in the redundancy management algorithms presented in Chapter 3.

The generation and propagation of runt pulses under the random pulse train and random wave train fault models can be illustrated by a simple example. Consider the synchronous digital system shown in Fig. 4.6. The outputs of the processing module are protected by an error-detecting code, and checked by an error detector. These modules are synchronized using a two-phase clock. At φ_1 a

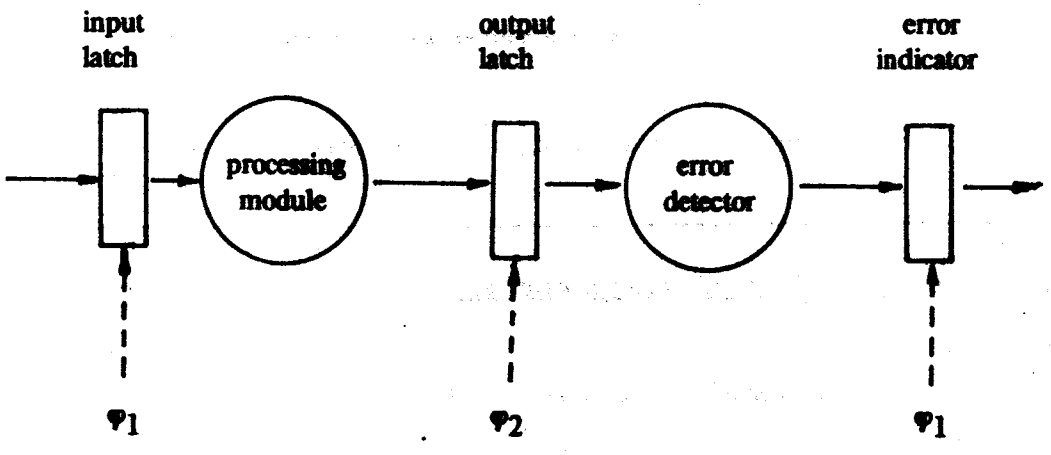


Fig. 4.6. An error detection scheme for synchronous systems.

new input for the processing module is stored in its input latch. The corresponding output is stored in its output latch at φ_2. This output is checked for errors and the error indicator flag is set at the immediately following φ_1. Suppose the processing module has failed and generates random wave trains or random pulse trains at its output port, violating the setup and holding time requirements of its output latch for synchronization with φ_2. Under these conditions the output latch of the processing module may enter a metastable state, as explained in Section 1.2.3, remain in this state for an arbitrarily long time, and then exit into a stable state that violates the error-detecting code and hence is erroneous. If this transition occurs late in the clock cycle following φ_2, the output of the error detector may not indicate an error at the next φ_1. In particular, the output signal of the error detector may still be residing at an immediate value between 0 and 1 when the next φ_1 occurs, and can be interpreted as indicating an absence of errors when latched into the indicator flag. In this way an erroneous output may be propagated from the faulty processing module to other hardware modules. This sort of pathological behavior cannot occur under the stuck-at fault model. Redundancy techniques for eliminating such pathological behavior in redundant packet communication systems are discussed in the next chapter.

Regarding packet communication protocols, we have presented a class of asynchronous protocols in Section 4.1 that is sufficiently general for implementing packet communication systems, and is yet still quite easy to characterize. It is possible to construct more general classes of protocols but their implementation may require more powerful and more complex encoding and decoding equipment. Separable sets can also be used to generalize the notion of an acknowledgment, to construct systems in which packets are transmitted in both directions over a single channel. The applicability of these generalizations remains to be investigated. It would also be of interest to compute the maximum number of distinct packets that can be represented using separable sets of binary strings of a given length. This problem has been studied in [5].

5. Control Module and Synchronizing Merge Module Design

In this chapter we study hardware implementation of control modules and synchronizing merge modules. While p-modules other than synchronizing merge modules can be implemented under unbounded gate delay assumptions [4], [31], [33], [39], we have found it convenient to design synchronizers, decoders and synchronizing merge modules using more conventional asynchronous state machine design techniques. Delay elements are inserted on selected paths in each control module to ensure that inputs to these asynchronous state machines indeed conform to assumptions made on their rate of change. Upper bounds on signal propagation delays among synchronizing merge modules in a redundant module are also used to implement time-out mechanisms in the consistency maintenance algorithm. Note that all such timing considerations are confined to within the same redundant module, and hence each redundant module can still be constructed without detailed knowledge of the internal operating speed of its neighbors, other than upper bounds on phase differences among packets and acknowledgments in the same batch received by the redundant module from these neighbors.

Hardware implementation of control modules and synchronizing merge modules is studied in Sections 5.1, 5.2 and 5.3, using the fault models given in Section 4.2. In Section 5.1 we study implementation of the timing synchronization algorithm in the synchronizer section of control modules. In Section 5.2 we examine techniques for decoding packet contents in the decoder section of synchronizing decoders. General implementation techniques for these decoders are discussed, without going into detailed case studies based on specific encoding schemes or decoder designs. In Section 5.3 we study implementation of the consistency maintenance algorithm in synchronizing merge modules. Our approach to constructing redundant packet communication modules and the implementation techniques explained in this chapter are illustrated by two design examples in Section 5.4.

Throughout Section 5.1, 5.2 and 5.3, we assume that there are 3f+1 byte slices in each redundant module, up to f of which may fail. There will thus be enough failure-independent byte slices to maintain synchronization and consistency using the algorithms presented in Chapter 3 under the abstract fault model presented therein. Hardware implementation of these algorithms, and their use in maintaining timing synchronization and consistency in packet communication systems subjected to failures modeled as in Section 4.2, are presented under this assumption. Application of our redundancy techniques to hardware modules which are not byte-sliced internally is illustrated in the design examples given in Section 5.4.

5.1 Synchronizer Implementation

Let us briefly review the operational environment of a synchronizer. Referring to Fig. 2.3, a synchronizing decoder receives input bytes from a synchronization set of 3f+1 fanout modules. Each byte is delivered by a fault-free fanout module as a packet to the synchronizing decoder using the adopted packet communication protocol. The arrival of a new byte at the synchronizing decoder is detected by a byte detector (Fig. 3.1a) and signaled to its synchronizer. In a fanout module, a synchronizer receives acknowledgment signals from its neighboring synchronizing decoders directly. The algorithm to be implemented in the synchronizer is described in Section 3.1. For every batch of input signals received, a synchronizer exchanges synchronization signals with other synchronizers, and then generates an output signal.

In the control modules environment, due to the use of handshake protocols, there is a feedback path from the output of the synchronizer to its signal sources, such that a fault-free input signal source S of the synchronizer will not send another signal to the synchronizer until the synchronizer has generated an output signal in response to the previous signal generated by S. A delay element can be inserted in this feedback path, in the control module containing the synchronizer, to regulate

the rate at which these input signal sources generate signals to the synchronizer. As we shall see in Section 5.1.1, this provides a convenient mechanism for ensuring the proper operation of asynchronous state machines used in the synchronizer.

For the synchronizer implementations studied in this section, a signal is implemented as a signal transition, either $0 \rightarrow 1$ or $1 \rightarrow 0$, on the corresponding wire. We furthermore adopt the convention that input signals in the same batch, and the output signal generated by the synchronizer in response to that batch, are all represented by signal transitions in the same direction. In the following discussion we need to distinguish between the logical signals received and sent by a module and the signal waveforms carried on its input and output wires. We call the former *logical signals* and use the term *signal* to denote the latter.

For each batch of logical input signals received by a synchronizer, at least $2f+1$ of them will be signal transitions generated by fault-free byte slices, and will be in-phase within some known δ. Failed slices can exhibit pathological behavior, and we will only consider the sort of pathological behavior allowed in our fault models. Under these input conditions, a hardware implementation of a synchronizer must support (S1), (S2) given in Section 3.1, and (S3):

(S1) For each new batch of input signals received, all fault-free synchronizers in the same synchronization set will generate output signals within a fixed time interval of each other. The duration of this interval is bounded by a constant which can be calculated from the time it takes a synchronizer to perform basic operations, and each basic operation can be performed in a fixed time.

(S2) If input signals in the same batch are generated by fault-free neighboring slices within a fixed known time interval of each other, then for each new input batch, a synchronizer will generate its output signal only after it has received all signals in that batch generated by fault-free slices.

(S3) The output signal generated by a fault-free synchronizer must not contain hazards or runt pulses.

The last property (S3) is especially important in self-timed hardware systems whose modules react to signal transitions on their input wires. All these three properties must be satisfied *together* in a synchronizer implementation to support our overall approach to timing synchronization. We first present a synchronizer design which implements the synchronization algorithm under *fault-free* operation, and then examine how this design can be enhanced to deal with hardware failures modeled by the three fault models explained in Section 4.2.

5.1.1 Synchronizer Implementation under Fault-Free Conditions

The basic operations in a synchronizer are to generate logical synchronization signals and logical output signals in response to the receipt of logical input signals or logical synchronization signals. These basic operations have the form:

"Receive logical signals on n of the 3f+1 input lines, then generate a logical signal."

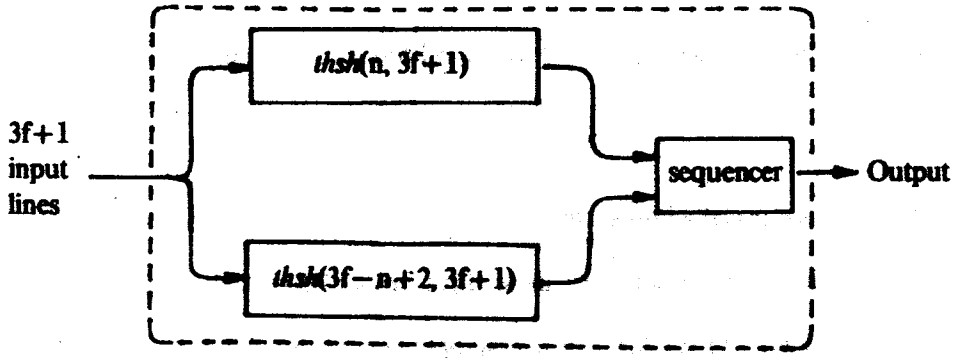
In an implementation, the logical signals received and generated by each such operation are all represented by signal transitions in the same direction. We first design a *generator* circuit (Fig. 5.1a), denoted by *gen*(n, 3f+1), for performing this operation. During fault-free operation, the input and output signals of the generator circuit undergo the following cycle:

(g1) All fault-free inputs are at 0.

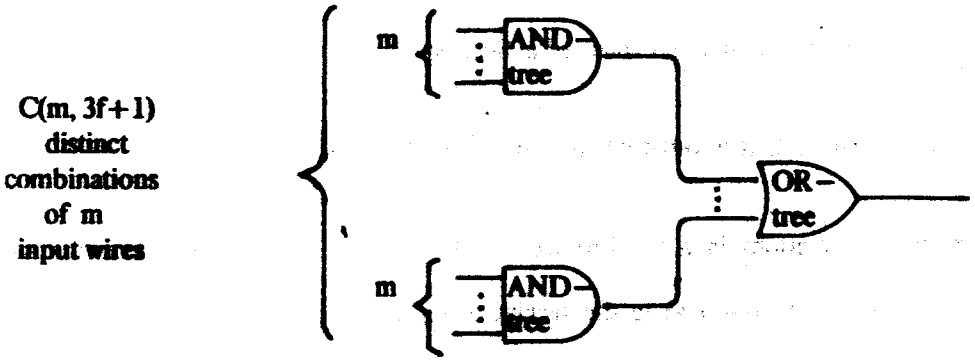
(g2) n or more of the inputs become 1, output is then set to 1. This corresponds to receiving a batch of logical input signals and generating a logical output signal in the synchronization algorithm.

(g3) All fault-free inputs are at 1.

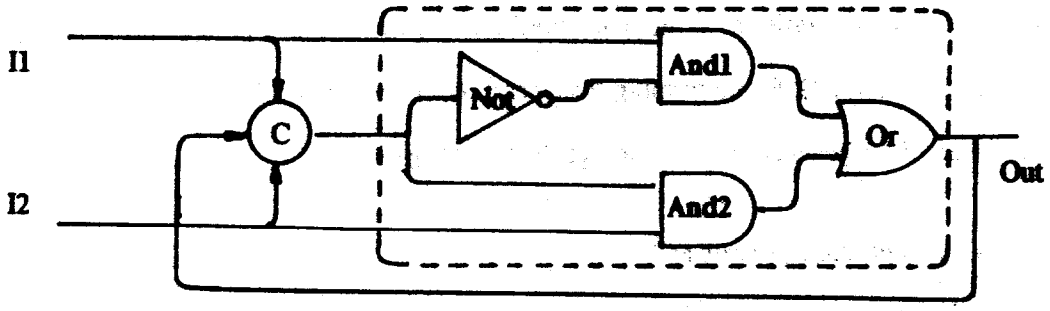
(g4) n or more of the inputs become 0, the output is then set to 0. This corresponds to



(a) A generator circuit *gen*(n, 3f+1).



(b) A threshold circuit *thsh*(m, 3f+1).



Demultiplexor

(c) A sequencer module.

Fig. 5.1. Hardware implementation of a generator circuit.

receiving a second batch of logical input signals and generating a second logical output signal in the synchronization algorithm.

In the context of control modules, due to the use of handshake protocols for packet communication, we can also control the duration at which the input signals to the synchronizer remain at input state g1 or g3. This can be achieved by delaying the acknowledgment returned by the *control module containing the synchronizer* to its predecessor. This kind of control is important because several asynchronous state machines are used in the synchronizer implementation, and their outputs would be hazard-free only if their internal state variables are given sufficient time to stabilize in between input changes. The rate of input change is controlled by holding the input states of a generator in g1 and g3 for as long as is necessary.

A generator circuit *gen*(n, 3f+1) (Fig. 5.1a) is constructed using two *threshold circuits* and an asynchronous state machine which we will call a *sequencer module.*

A threshold circuit (Fig. 5.1b), denoted by *thsh* (m, 3f+1), is used to detect the arrival of m logical signals. It is constructed from C(m, 3f+1)[1] trees of AND gates and a tree of OR gates. Its output is 1 if and only if m or more of its 3f+1 input signals are at 1, and is free of hazards under the following operating conditions:

As the input state to a threshold circuit cycles from g1 through g4, a fault-free input signal will not make another transition until after the output signal transition generated in response to the previous transition on that input signal has been observed on the output line of the threshold circuit.

---

1. C(m, 3f+1) is the number of combinations of choosing m objects out of 3f+1 distinct ones.

Since these threshold circuits are used inside control modules, this condition is automatically satisfied by using asynchronous handshake protocols for packet communication in the hardware system.

In the generator (Fig. 5.1a), the *thsh*(n, 3f+1) circuit detects the input state transitions from g1 to g2, the *thsh*(3f−n+2, 3f+1) circuit detects the input state transitions from g3 to g4. The output signals of these two threshold elements cycle through the following states:

(t1) *thsh*(n, 3f+1) = 0, *thsh*(3f−n+2, 3f+1) = 0, (generator inputs at g1)

(t2) *thsh*(n, 3f+1) = 1, *thsh*(3f−n+2, 3f+1) = −, (generator inputs changes from g1 to g2)

(t3) *thsh*(n, 3f+1) = 1, *thsh*(3f−n+2, 3f+1) = 1, (generator inputs at g3)

(t4) *thsh*(n, 3f+1) = −, *thsh*(3f−n+2, 3f+1) = 0, (generator inputs changes from g3 to g4)

and then back to t1 when the input signals to the generator change from g4 back to g1.

Under the signal representation conventions, the output signal of the generator should go from 0 to 1 when the input signals of the generator go from g1 to g2 or, equivalently, when the output signal of the threshold element *thsh*(n, 3f+1) goes from 0 to 1. Similarly the output signal of the generator undergoes a 1 to 0 transition upon a g3 to g4 transition on the input signals of the generator or, equivalently, a 1 to 0 transition on the output signal of the threshold element *thsh*(3f−n+2, 3f+1).

The output of the generator circuit is derived from the outputs of the two threshold circuits using the asynchronous state machine, which we will call a *sequencer module*, shown in Fig. 5.1c. The sequencer module has a single binary state variable, implemented with a C-element. When both inputs I1 and I2 are 0, the output of the C-element is 0 and the next 0 → 1 transition on I1 is transmitted to the output line *Out*. When both inputs are 1, the output of the C-element is 1, and the next 1 → 0 transition on I2 is transmitted to *Out*. The output signal delivered at *Out* by a sequencer

module is hazard-free under a delay[1] assumption and an input assumption:

(1) If both input signals to the demultiplexor (Fig. 5.1c) have the same value as its output signal, then selecting one input signal instead of the other by changing the output of the C-element will not cause a spurious pulse at *Out*. This can be assured if in the demultiplexor (Fig. 5.1c),

delay (Not) + delay (And1) > delay (And2) + delay (Or)

(2) The input states remain at 00 and 11 long enough to allow the internal state of the sequencer module to settle down before the next input change occurs. The previously discussed mechanism of delaying the return of acknowledgments from a control module to its predecessor is useful for ensuring that this condition is satisfied. By holding the input signals of the generator circuit at g1 and g3, the output signals of the two threshold elements, and hence the input signals to the sequencer module, can be held at t1 (00) and t3 (11), respectively.

□

Three generator circuits are used in a synchronizer (Fig. 5.2a):

G1: *gen* (f+1, 3f+1), generates a logical signal after receiving f+1 logical input signals,

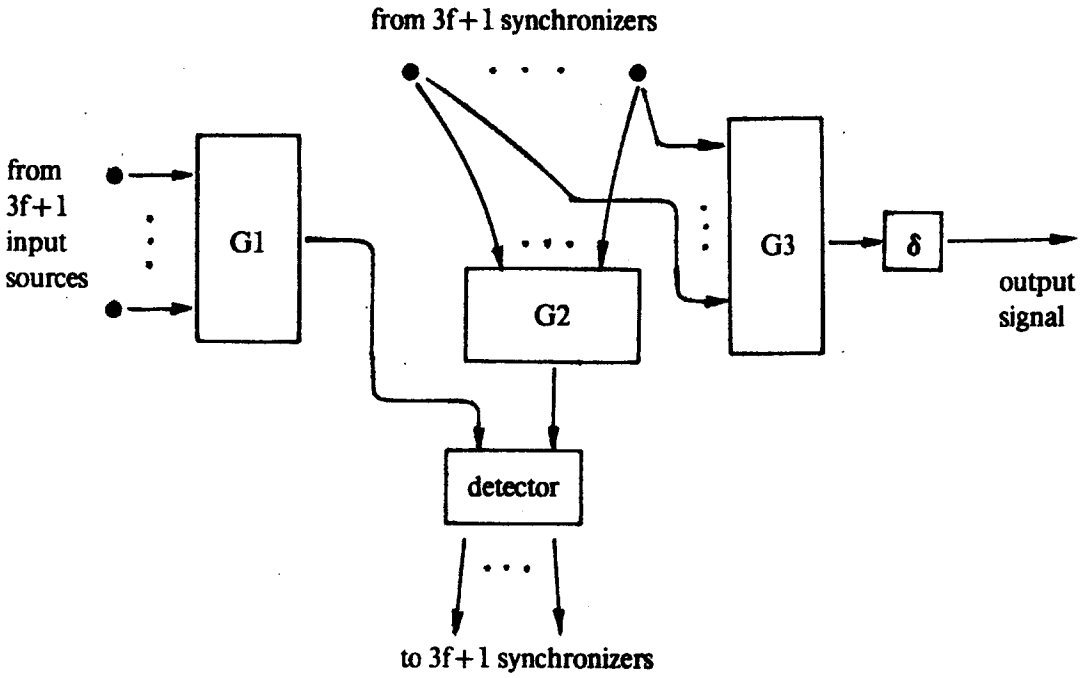
G2: *gen* (f+1, 3f+1), generates a logical signal after receiving f+1 logical synchronization signals,

G3: *gen* (2f+1, 3f+1), generates the logical synchronizer output signal after receiving 2f+1 logical synchronization signals.

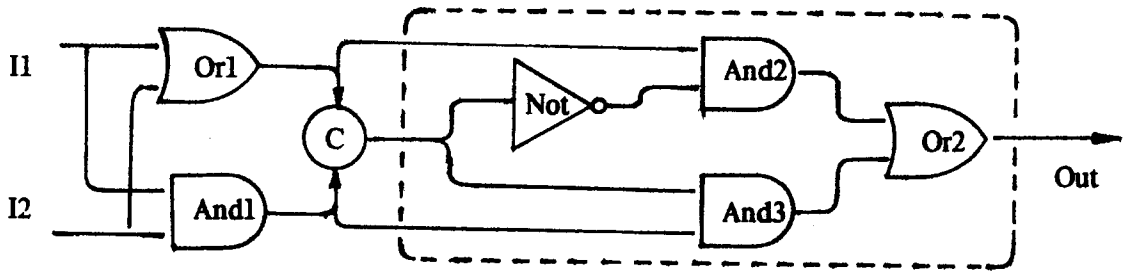
Logically the synchronizer generates a synchronization signal as soon as either G1 or G2 generates a logical signal. Under our signal representation, this is implemented by combining the output signals generated by G1 and G2 using another asynchronous machine called the *detector*

---

1. For delay considerations, a logic gate or a C-element is modeled as a hardware element, which reacts instantaneously to input changes according to the corresponding boolean function or transition diagram (Fig. 4.2b), followed by a delay element.



(a) Synchronizer design.



Demultiplexor

(b) A detector module.

Fig. 5.2. Hardware implementation of a synchronizer.

module (Fig. 5.2b). A detector module differs from a sequencer module in that it does not pass a 0 → 1 transition on one input signal and then a 1 → 0 transition on the other to its output line. The detector module also has a single binary state variable, implemented with a C-element. When both inputs I1 and I2 are 0, the output of the C-element is 0, and the next 0 → 1 transition on *either* I1 or I2 is transmitted to the output line *Out*. When both inputs are 1, the output of the C-element is 1, and the next 1 → 0 transition on *either* I1 or I2 is transmitted to *Out*. The output signal delivered at *Out* by a detector module is hazard-free under a delay assumptions and an input assumption:

(1) If both input signals to the demultiplexor have the same value as its output signal, then selecting one input signal instead of the other by changing the value of "select" from 0 to 1 or from 1 t 0 will not cause a spurious pulse at *Out*. This can be assured if in the demultiplexor (Fig. 5.2b):

delay(Not) + delay(And2) > delay(And3) + delay(Or2)

(2) The input states remain at 00 and 11 long enough to allow the internal state of the detector module to settle down before the next input change occurs. This condition can be satisfied by preventing input signals and synchronization signals from fault-free modules from changing too rapidly, which can again be achieved by delaying the logical acknowledgment signals returned by the control module containing the synchronizer to its predecessor by an appropriate amount of time.

□

Finally, to satisfy (S2), the output of G3 is delayed δ seconds (Fig. 5.2a), where δ is the phase difference among logical input signals in the same batch delivered to the synchronizer.

This completes the description of our synchronizer design. Let us examine the operational characteristics of this design during fault-free operation and the support it provides for our overall approach to constructing fault-tolerant packet systems in more detail. We have noted that a

synchronizer implementation must satisfy properties (S1), (S2) and (S3) together. (S1) is satisfied since the basic operations in the synchronizer are implemented using hardware elements such as C-elements and logic gates, each of which has a fixed, bounded, reaction time under all possible input conditions presented to them in the synchronizer during fault-free operation. (S2) is satisfied by delaying the output of G3 by δ. (S3) is achieved by designing the threshold circuits, the sequencer modules and the detector module to be hazard-free under these input conditions. For system integration, the phase difference among packets and acknowledgments in the same batch delivered to a control module is used to determine the proper delay element for delaying the logical output signals generated by G3.

To analyze the fault tolerance capability of the synchronizer implementation, let us define an *operation cycle* of a synchronizer as the period between the time when all fault-free input signals to the synchronizer are at the same logic level to the next time when all these fault-free input signals have switched to the complementary logic level. Due to the use of asynchronous packet communication protocols, operation cycles can vary arbitrarily in length. Proper synchronization assures only that all fault-free input signals will change state together within the last δ time units of each operation cycle, for some fixed δ. We can characterize the restriction on input signals delivered by failed modules imposed by the abstract fault model presented in Section 3.1 as:

> Suppose in an operation cycle every fault-free input signal from a group of neighboring control modules, or from a group of synchronizers in the same synchronization set, makes a transition from one logic value a to the other logic value $\sim a$. Then an input signal, generated by a failed module in that group, either stays at a for the entire operation cycle, or makes the same transition exactly once and then stays at $\sim a$ in that cycle.

Under this restriction on input signals delivered by failed modules, (S1), (S2) and (S3) can be maintained using the synchronizer implementation presented above. We will show that under the stuck-at fault model, the output signals generated by failed modules indeed obey this restriction, and hence the above synchronizer can be used directly to maintain timing synchronization under this fault model. Under the random pulse train fault model, we show that we can *filter* the output signals generated by failed modules to derive signals which obey this restriction, and then feed these signals to the synchronizer implementation presented above to maintain timing synchronization. An approach to tolerating failures under the random wave train fault model is discussed in Section 5.1.4.

5.1.2 Synchronizer Implementation under the Stuck-At Fault Model

A synchronizer in a fanout module receives acknowledgments directly from its neighboring slices and will stop receiving acknowledgments on any input line which is stuck at either 0 or 1. Thus a neighboring slice which has failed will appear to a synchronizer in a fanout module as if it has stopped sending acknowledgments.

A synchronizer in a synchronizing decoder receives signal transitions from its neighboring slices indirectly through byte detectors monitoring the output packet bundles of these slices. Suppose a failed byte slice attempts to set its output packet bundle to a certain state to transmit a new byte. If some wires in its output packet bundle are stuck at either 0 or 1, then states which may appear on the bundle are restricted to those with the corresponding bits fixed at the corresponding values. If the failed byte slice attempts to set its output packet bundle to a state which is not in this restricted subset, the desired transition will not be observed by the byte detector monitoring the packet bundle and will not cause a signal transition on the output of the byte detector. Later on, the failed byte slice may be successful in delivering another packet encoded by a packet bundle state in the restricted subset. The arrival of this new packet is then detected by the byte detector and signaled to the

synchronizer. The stuck-at fault model for packet communication modules (Fig. 4.3) also assumes that a failed slice behaves just like the fault-free slice, except that some of its output lines are stuck at either 0 or 1. Thus a failed slice will attempt to deliver a new byte, using the adopted protocol, only when all fault-free slices in the same redundant module are prepared to do so within a fixed interval δ.

These fault phenomena can be illustrated using the byte detector designed to receive packets in dual-rail (Fig. 4.2). If both lines in an input wire pair to the byte detector are stuck, at either 0 or 1, or one of the wires in a wire pair is stuck at 1, the output of the byte detector will also become stuck at some logical value, either 0 or 1. If one of the wires becomes stuck at 0, packet communication can continue until a dual-rail bit encoded by a 1 on the wire suffering the stuck-at fault and a 0 on the other wire in the wire pair is transmitted. The wire pair will then stay in the spacer state, and the byte detector will not signal the arrival of the new byte. The corresponding bit of a subsequent packet may, however, be encoded by a 0 on the wire stuck at 0, and a 1 on the other wire in the wire pair. The arrival of this new packet is then be detected by the byte detector and signaled to the synchronizer.

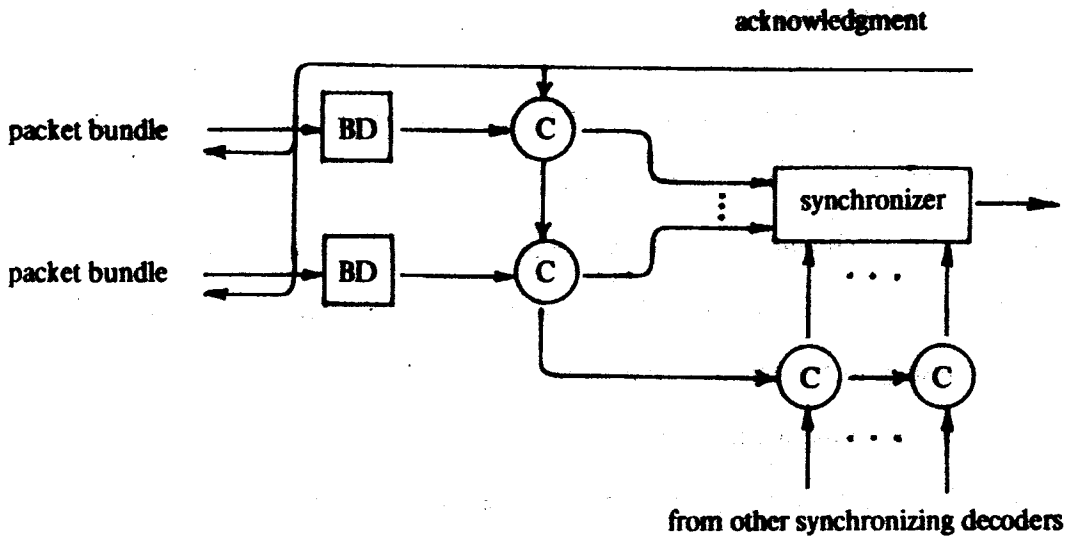
Thus under the stuck-at fault model, the input signals delivered to a synchronizer satisfy the restriction explained at the end of Section 5.1.1. The synchronizer design presented in Section 5.1.1 can thus be used to construct control modules which can maintain properties (S1), (S2) and (S3) in a synchronization set.

5.1.3 Synchronizer Implementation under the Random Pulse Train Fault Model

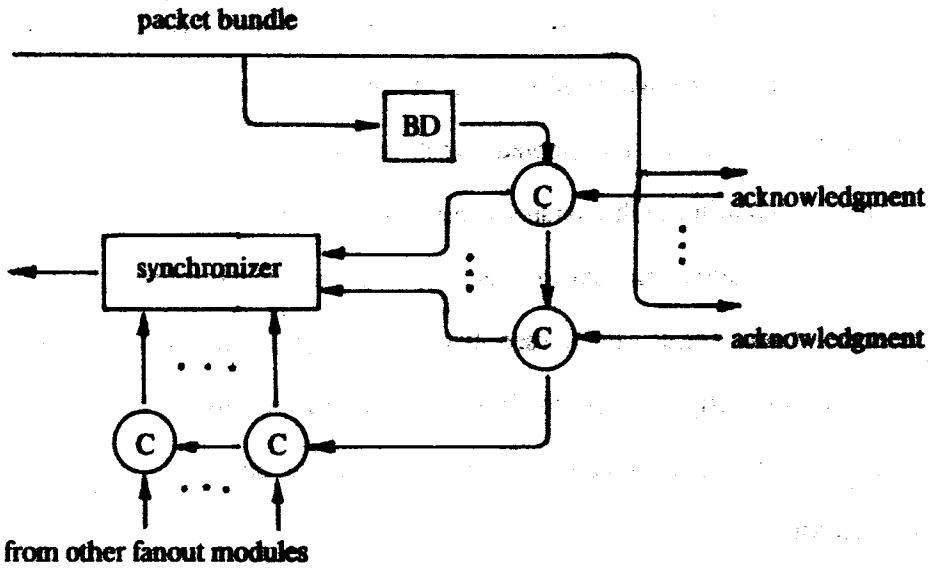
Under the random pulse train fault model (Fig. 4.5), a random pulse train signal generated by a failed byte slice neighbor can propagate through a byte detector and appear on the corresponding input line of the synchronizer. If this signal is processed by the synchronizer presented in Section 5.1.1 directly, property S3 can be violated and the synchronizer can deliver random pulse train outputs, according to the assumed fault model for interaction between random pulse trains and hardware elements in the synchronizer. Under this fault model, however, a random pulse train can be "filtered" by passing it through a C-element whose output is regulated by a fault-free reference signal (Fig. 4.5c).

In a control module, we can derive a reference signal for this purpose from either an acknowledgment bundle (in synchronizing decoders) or a packet bundle (in fanout modules) in the control module containing the synchronizer. For synchronizer design we can assume that these signals are free of errors and obey the adopted asynchronous handshake protocol. This is because a byte slice is taken as a unit for fault tolerance considerations and in a fault-free slice synchronizing decoders and fanout modules receive acknowledgments and packets, respectively, from the p-module in the same slice. These signals can be used as reference signals because for the class of packet protocols presented in Section 4.1, acknowledgments and packets always alternate in a channel operated under fault-free conditions. We next illustrate the application of these ideas in a synchronizing decoder.

For a synchronizing decoder, our strategy is to filter the signals generated by the byte detectors and by other synchronizers in the same synchronization set with a column of C-elements (Fig. 5.3a) regulated by the acknowledgment signal received from the p-module in the same slice, before feeding these signals to the synchronizer. Conceptually we are reinforcing the packet protocol at the



(a) C-element filters in synchronizing decoders.



(b) C-element filters in fanout modules.

Fig. 5.3. Filtering random pulse train inputs in control modules.

C-element filters such that the arrival of new bytes and logical synchronization signals at the synchronizer alternate with the receipt of logical acknowledgment signals from the p-module. In terms of processing binary signals, we are imposing the following restriction on the input signals received by a synchronizer:

> Suppose in an operation cycle every fault-free input signal in a group of neighboring control modules, or in a group of synchronizers in the same synchronization set, makes a transition from one logic value a to the other logic value $\sim a$. Then an input signal belonging to that group, generated by a failed module, either stays at a for the entire operation cycle, or makes the same transition exactly once and then stays at $\sim a$ in that cycle. All such transitions can occur only after the logical acknowledgment signal for the previous batch of transitions has been received by the synchronizing decoder containing the synchronizer.

Similarly, a reference signal can be derived from the packet bundle in a fanout module (Fig. 5.3b) and used to filter random pulse trains on acknowledgment lines before feeding them to the synchronizer in the fanout module.

Any input signal obeying this restriction also obeys the restriction given at the end of Section 5.1.1. A synchronization set of control modules can be constructed using synchronizers enhanced by these C-element filters, and properties (S1), (S2) and (S3) can be maintained in this synchronization set under the random pulse train fault model.

5.1.4 Synchronizer Implementation under the Random Wave Train Fault Model

As illustrated in Fig. 4.6, random wave trains can propagate through hardware elements in the synchronizer and cause runt pulses on the output signals of threshold circuits, sequencers and detectors, as well as on the output signal of the synchronizer itself. Random wave trains are basically analog signals and can be filtered with analog filters. Since sequencers and detectors are asynchronous state machines, it is attractive to filter the output signals of *threshold circuits* before feeding them to these two types of modules. We next outline an approach to enhance the synchronizer design presented in Section 5.1.1 to deal with random wave trains. Its detailed verification requires analog circuit and signal analysis and is beyond the scope of this thesis.

When all fault-free input signals to a threshold circuit are at the same signal level a, the output of the threshold circuit will also be set at a. As soon as one fault-free input signal changes its state to $\sim a$, it becomes possible for the faulty signals to conspire together and cause the output of the threshold circuit to oscillate between 0 and 1 and generate runt pulses. If input signal transitions from fault-free sources are in-phase within α, the time period during which oscillation may occur and the width of runt pulses will both be bounded by α. When α is known, we can build a lowpass filter to filter out these oscillations and runt pulses.

Let us first consider adding lowpass filters to each of the generator circuits in the synchronizer. For the threshold elements in the generator circuit G1 (Fig. 5.2a) their logical input signals are in-phase within δ, where δ is known when specifications of neighboring redundant modules are given. The lowpass filters on their output signals should thus be designed to filter out all oscillations with period shorter than δ. For generator circuit G2, a little thought reveals that all fault-free synchronizers are guaranteed to generate logical synchronizer signals in the same batch within δ, since they will all receive at least f+1 logical input signals from fault-free input sources within δ.

The output signals of the threshold circuits in G2 should thus also be filtered to eliminate oscillations with period less than δ using lowpass filters. For generator circuit G3, we showed in Lemma 3.1 that fault-free synchronizers will generate logical synchronizer signals in the same batch within 2ρ, where ρ is conceptually an upper bound on the time it takes for a synchronizer to generate a logical synchronizer signal after receiving at least $f+1$ logical synchronizer signals, plus the time it takes for this newly generated signal to reach the other synchronizers. In a synchronizer implementation, ρ is an upper bound on the delay through the generator circuit G2 and the detector module (Fig. 5.2), plus an upper bound on the propagation delay between synchronizers.

To support our timing synchronization methodology, a synchronizer implementation must satisfy properties (S1), (S2) and (S3) given at the beginning of this section. (S2) does not concern the design of lowpass filters. To satisfy (S1), a signal transition, from either $0 \rightarrow 1$ or from $1 \rightarrow 0$, must be propagated through the lowpass filter in a fixed time interval. Consider when the output signal of a threshold circuit used in the above contexts is at 0. Immediately before changing to 1, this signal may oscillate between 0 and 1, for up to α seconds, for some fixed α. We require that as soon as this signal "stabilizes" at 1, a $0 \rightarrow 1$ transition will be observed on the output signal of the lowpass filter within β seconds, where β is another fixed constant independent of α. This requirement also holds for $1 \rightarrow 0$ transitions. To satisfy (S3), the output signal of the lowpass filter should not contain runt pulses.

Implementation of lowpass filters in available hardware technology, deviations exhibited by these implementations from the desired characteristics, and consequences of such deviations in maintaining timing synchronization must be investigated before the effectiveness of this approach can be evaluated.

5.2 Decoding Section Implementation

In our approach to constructing redundant packet communication systems, hardware functions are implemented with byte-sliced modules whose output words and acknowledgments are *encoded*. A word is encoded in that it consists of a batch of packets, each generated by a separate byte slice and containing one byte of the word, and the separable sets used in the packet protocols to deliver each batch *jointly* support some error-detecting or error-correcting capability. The example of even-parity dual-rail protocol has been given at the end of Section 4.1 to illustrate such protocols. An acknowledgment generated by a byte-sliced module is encoded in that it is represented by a batch of acknowledgment signals, one generated by each byte slice. Hardware failures, limited to some maximum number of byte slices prescribed by the error-detecting or error-correcting capabilities of the protocol, are detected and/or masked by decoding output words and acknowledgments generated by a byte-sliced module. The goal of maintaining timing synchronization and consistency in a redundant system is to provide an accommodating environment for applying encoding techniques. As we have explained in previous discussions, the fault tolerance capabilities of an encoding scheme can be compromised if either timing synchronization or consistency is not maintained. Only those failures whose occurrences do not lead to loss of synchronization or consistency can be detected and/or masked reliably in our approach.

In the last section we have discussed the hardware implementation of fault-tolerant synchronizers which can be used to maintain timing synchronization under the stuck-at fault model, the random pulse train fault model and the random wave train fault model. Organization of decoding sections are studied in this section. We note that the problem of "decoding" an acknowledgment in a fanout module (Fig. 3.1b) is taken care of using fault-tolerant synchronizers. We thus only need to deal with the problem of decoding input words in synchronizing decoders. We will explain several general techniques to deal with random pulse train and random wave train input

signals in decoding sections, but will not present decoder designs for specific error-detecting or error-correcting protocols.

A synchronizing decoder is organized internally into byte detectors, a synchronizer and a decoding section (Fig. 3.1a). Input words are delivered to a decoding section on several subbundles, some of which may be connected to failed byte slices. A word is encoded to support either error detection or error correction. For error detection, input signals to the decoding section are monitored to determine whether the input word received contains any errors. To limit error propagation, erroneous input words are "held up" in the decoding section instead of being forwarded to p-modules. For error correction, input signals to a decoding section are used to regenerate the error-free word, which is subsequently forwarded to the p-module in that slice.

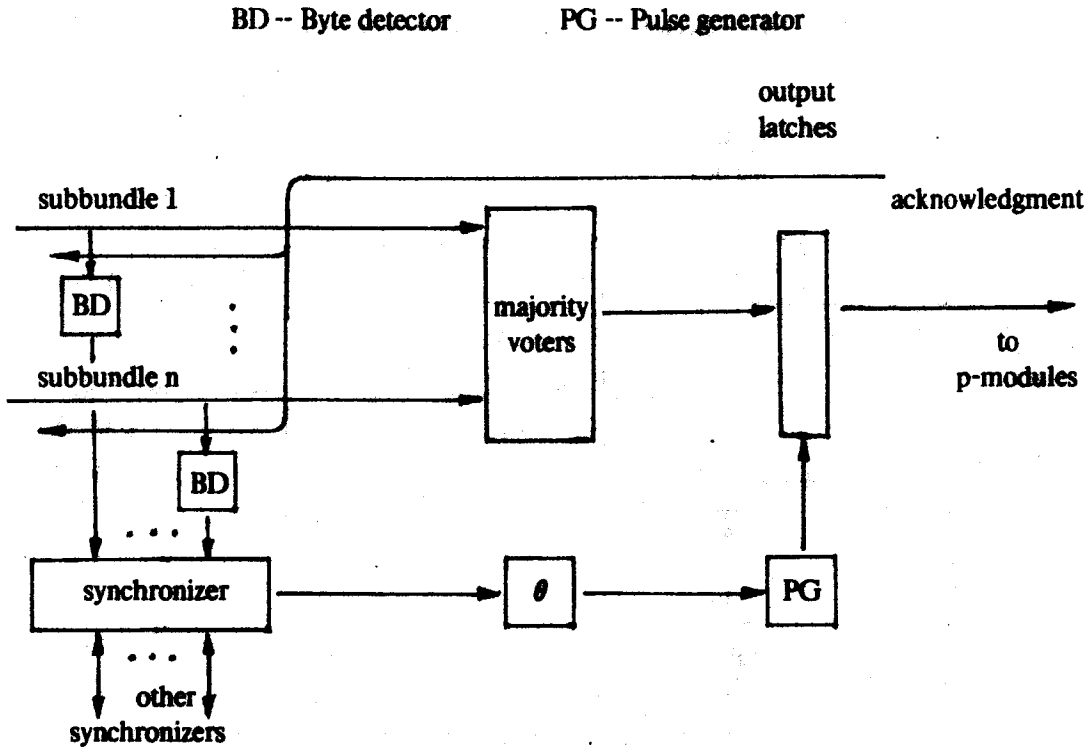
In a decoding section input signals are decoded by *signal decoders*. We restrict our attention to signal decoder designs *which can generate error signals or regenerate the error-free input word within some fixed time interval after all of its fault-free input signals have stabilized.* Using signal decoders with this property, our implementation strategy for a decoding section is to receive a timing signal from the synchronizer, which indicates that all fault-free input signals to the decoding section have stabilized, wait a fixed time interval for the output signals of the signal decoder to stabilize, and then use these output signals in error detection and/or correction.

Let us examine the requirements imposed on signal decoder designs more carefully before studying implementation issues. When all fault-free input signals to a signal decoder have stabilized, input signals from failed neighboring slices can exhibit pathological behavior according to the stuck-at fault model, the random pulse train fault model or the random wave train fault model. We require that within a fixed time interval after all fault-free input signal values are available, the output signals of the signal decoder must have stabilized at their appropriate values. Oscillation and runt

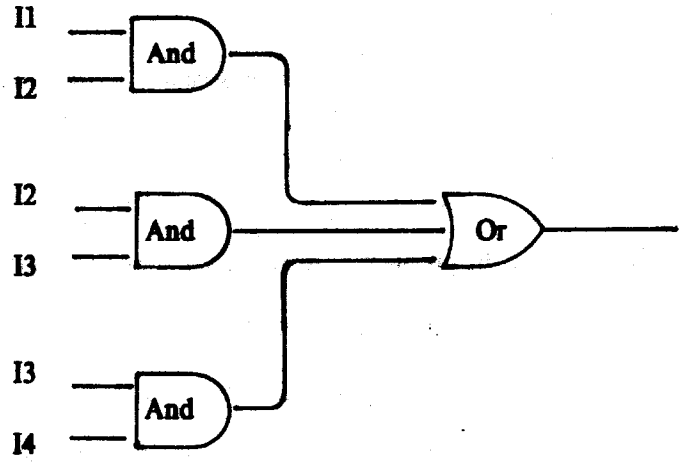
pulses are permitted, however, in the interim. Under these requirements, signal decoders must be designed so that runt pulses and oscillations cannot propagate through them if fault-free input signals have already stabilized for some time.

An encoding scheme for which we can design such a signal decoder is the familiar *byte replication* scheme. In this scheme each word consists of several identical bytes and the error-free word is derived using majority voting. A decoding section for a byte replication scheme (Fig. 5.4) designed according to our strategy consists of a signal decoder which contains a group of majority voters, a delay element θ to time the propagation delay through the signal decoder, and a column of output latches to store the output values of the majority voters after their output signals have stabilized. A majority voter is constructed out of AND and OR gates (Fig. 5.4b). Under our fault models, a majority voter has the characteristic that if two of its input signals are at the same value, its output signal will stabilize at that value after a finite time period, independent of whether the third input signal is stuck at some logical value, delivering a random pulse train, or delivering a random wave train.

Another approach to implement signal decoders with the desired property is to store input signals in a column of latches (Fig. 5.5) and decode the outputs of these latches after they have stabilized. In this arrangement the latches are activated by the output signal of the synchronizer. Non-faulty input signals and those stuck-at either 0 or 1 will always have stabilized when the latches are activated by the synchronizer output signal. A random pulse train or random wave train may cause its receiving latch to enter a metastable state, according to the fault models in Section 4.2. A latch is allowed β seconds (Fig. 5.5) to settle down, where β is a circuit technology parameter. The outputs of the input latches are then decoded and stored in a column of output latches, as in a replication scheme. In Fig. 5.5, α is an upper bound on the propagation delay through the signal decoder.



(a) Synchronizing decoder



(b) majority voter

Fig. 5.4. Synchronizing decoder design for replication schemes.

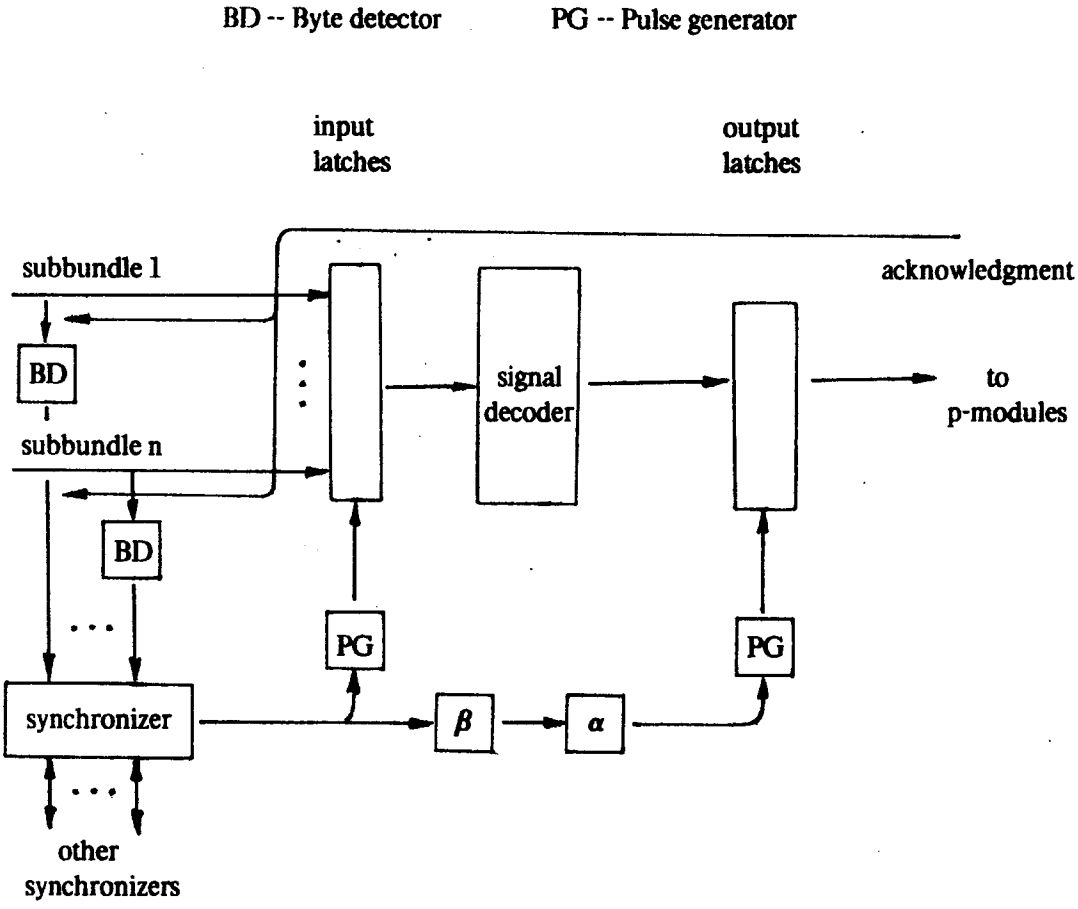


Fig. 5.5. A synchronizing decoder design for error correction.

This strategy of reducing the problem of receiving random pulse trains and random wave trains to that of waiting for latches to exit from their metastable states can also be used to design decoding sections for fault detection, as illustrated in Fig. 5.6. An input word to the decoding section is first latched into input latches. After β seconds, the outputs of the input latches are strobed into a column of *internal latches*. The contents of these internal latches are forwarded to the output latches of the decoding section only if they contain no errors. Suppose the adopted protocol is based on separable sets A and B. The key input property used for fault detection in the configuration shown in Fig. 5.6 is that if the next input word delivered to the detector should be encoded in one of the separable sets,

say by a_1 in A, then no hardware failures to be detected can cause a packet encoded in some other a_2 in A to be delivered to the detector. Otherwise a_2 will be accepted instead of a_1 and forwarded to the p-modules, defeating the fault detection scheme. As long as this property holds, errors can be confined and detected by monitoring the contents of the internal latches with a packet detector (Fig. 4.2a) designed to recognize the separable sets A and B. After the input word is stored in the internal latches, the packet detector monitoring the outputs of these internal latches will generate a signal transition on its output line only if the input word stored in the internal latches is encoded in the proper separable set, and hence free of errors. If the input word stored in the internal latches is erroneous, i.e., not encoded in the proper separable set, no transition will be delivered by the packet detector, and the erroneous word will not be stored into the output latches.

An alarm pulse is generated from the output of the synchronizer and the output of the packet detector (Fig. 5.6). The pulses generated from these two signals are of bounded width during fault-free operation. If an erroneous or unstable input state is stored in the internal latches, the output of the packet detector will not change state and the width of the alarm pulse becomes unbounded. This property is useful in designing alarm detectors.

This fault detector design in Fig. 5.6 is also applicable to detecting acknowledgment signal failures. The two separable states used on acknowledgment bundles are the two singletons { 111...111 }, and { 000...000 }. An AND gate is a receptor for the first set, an OR gate a receptor for the second.

Note that under the random pulse train and random wave train fault models pathological input conditions need not persist until detected. Such conditions may exist for a short time and then disappear before any fault detection mechanism has reacted to them. The fault detection scheme nonetheless assures, if only in a probabilistic sense due to metastable state phenomena, that no

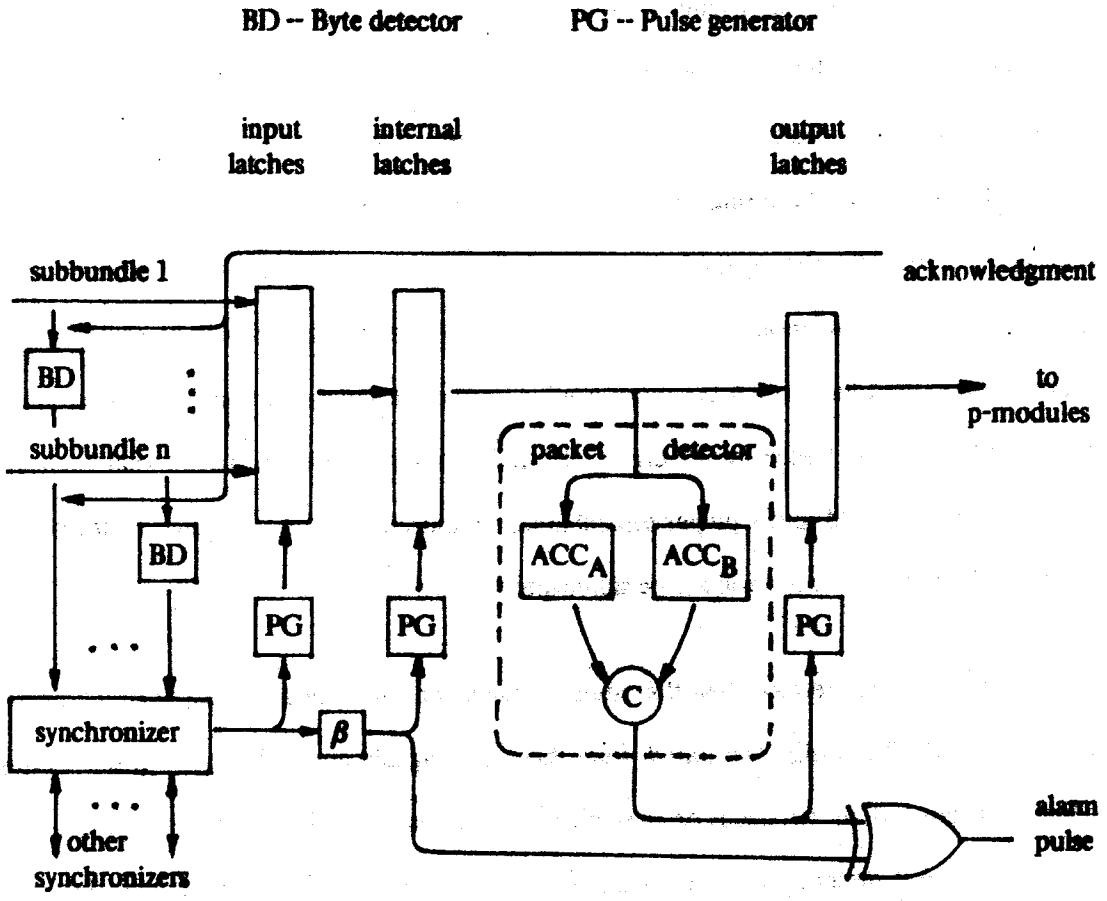


Fig. 5.6. A detector design for coding schemes.

---

erroneous packet will be delivered to the p-modules.

5.3 Implementation of the Synchronizing Merge Module

With the synchronizer design and decoding section design presented in Section 5.1 and 5.2, we have a complete methodology for incorporating redundant hardware in *determinate* packet communication systems to combat hardware failures. The synchronizing merge module is introduced to illustrate an extension of this methodology, to a class of *non-determinate* systems. The basic non-determinate operation we consider is that of merging two input streams into a single output

stream (Fig. 2.4). In a redundant system, this operation is performed in a *redundant merge* module using *synchronizing merge* modules as p-modules in its slices.

In a redundant merge module, every input word is decoded by a synchronizing decoder before being forwarded to a synchronizing merge module. A slice, however, forwards only one byte of each received word, so that a single byte slice failure affects at most one byte of any word delivered by the redundant merge module. To apply encoding techniques effectively, it is necessary to ensure that bytes from input words received at different input ports will not be "mixed up" together in a redundant merge module. In the terminology introduced in Section 2.1, it is necessary to ensure that bytes in the same batch generated by fault-free synchronizing merge modules in the same redundant merge module all belong to the same input word. The consistency problem is solved by allowing synchronizing merge modules in a redundant merge module to communicate with each other after receiving an input word, to jointly determine the input port from which every fault-free synchronizing merge module will forward the next output byte to the successor redundant module.

In Section 3.2, we have presented an algorithm for exchanging messages among synchronizing merge modules, so that even if failed modules can send different messages to different modules, fault-free synchronizing merge modules can still agree on which input port to service next. A hardware structure for synchronizing merge modules and implementation of the consistency maintenance algorithm are described in this section. We first present a hardware structure for implementing synchronizing merge modules and review the operation of a synchronizing merge module using this structure. Implementation of the consistency maintenance algorithm is then discussed in further detail and its fault tolerance capability explained. For maintaining timing synchronization, p-modules in a redundant module must be performance compatible. Performance compatibility among synchronization merge modules implemented using these techniques is also analyzed. As noted in Section 3.2, only the single failure case, in which four synchronizing merge

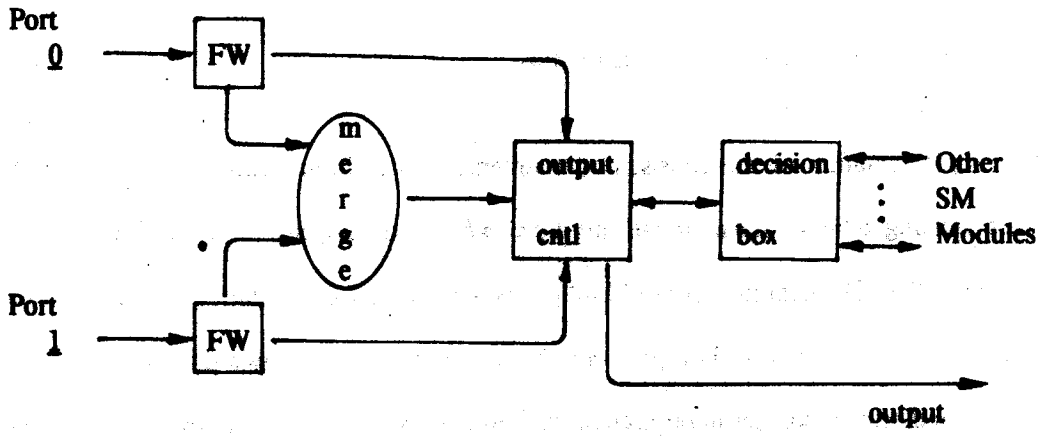
modules are used in a redundant merge module to tolerate hardware failures in any one of them, is considered.

5.3.1 Hardware Structure of a Synchronizing Merge Module

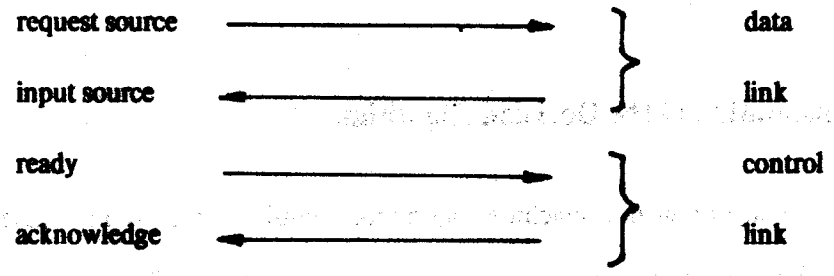
Packet flow at an input port of a synchronizing merge module is regulated using a *forward and wait* (FW) module (Fig. 5.7a). A FW module forwards the input packets it has received directly to an *output cntl* module. For each packet received, the FW module also generates an output request tagged with the corresponding input port identifier (0 or 1). These output requests are merged together and then processed by *output cntl*. A FW module will accept a new packet only after it has received acknowledgments for both the preceding packet, from *output cntl*, and for the output request it has generated for the preceding packet, from the merge module.

A merge module receives input packets at its two input ports and forwards them at its output port as they arrive. When a conflict arises due to two input packets arriving within a short time interval, arbitration is performed and the two packets may be forwarded in either order.

When a new request arrives at *output cntl*, its tag (which will be referred to as the *request source*) is used to invoke the decision algorithm. The *input source* for the next output packet is selected jointly with other synchronizing merge modules under this decision algorithm. If the input source is different from the request source proposed by *output cntl*, the corresponding output request for the input source will be pending at the output of the merge module. This is because the input packet from the rejected request source has not yet been accepted and the corresponding FW module is designed not to process a new input packet from the input port it is guarding under the circumstances. Thus the output request from the input source will be the only one pending at the merge module, and will be forwarded to the *output cntl* module next. *Output cntl* absorbs this output request for the input source from the merge module, accepts an input word from the FW module



(a) Hardware organization.



(b) Communication protocol between *output cntl* and *decision box*.

Fig. 5.7. The Synchronizing Merge Module.

---

guarding the input source, extracts one byte from it, and forwards this byte to the successor redundant module via a fanout module. When *output cntl* receives the acknowledgment for this byte, it returns an acknowledgment to the FW module, and if it already has a request source that was rejected in the previous round of decision, it will immediately resubmit this request source. Otherwise it waits for a new request to arrive from the merge module before activating the decision algorithm again.

We will omit the implementation details for the FW module and the *output cntl* module since they do not illustrate new fault tolerance techniques.

For implementation of the consistency maintenance algorithm, request sources and input sources are exchanged between *output cntl* and *decision box* under a *generalized data link/control link protocol* (Fig. 5.7b). The data link/control link protocol is explained in Chapter 1 (Fig. 1.1). Under the generalized data link/control link protocol, the data link consists of two data wires for sending input sources and request sources in opposite directions (Fig. 5.7b). A new request source is available to *decision box* when a *ready* signal is sent on the control link. The corresponding input source is available on the other data wire when the corresponding *acknowledgment* signal is returned.

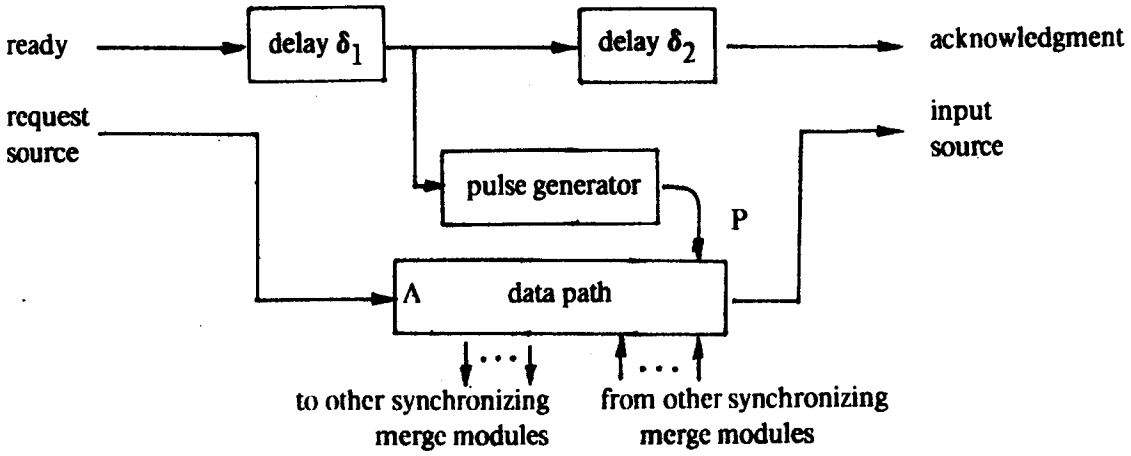
5.3.2 Implementation of the Decision Algorithm

The decision algorithm in a synchronizing merge module is invoked with either $\underline{0}$ or $\underline{1}$ as its request source. For every request source received from *output cntl* the following hardware operations are carried out, corresponding to Steps (i) through (v) of the decision algorithm presented in Section 3.2. The data path in synchronizing merge module A for carrying out these operations is shown in Fig. 5.8b. In this figure the symbol "X_{YZ}" denotes the request source of X as told to Z by Y.

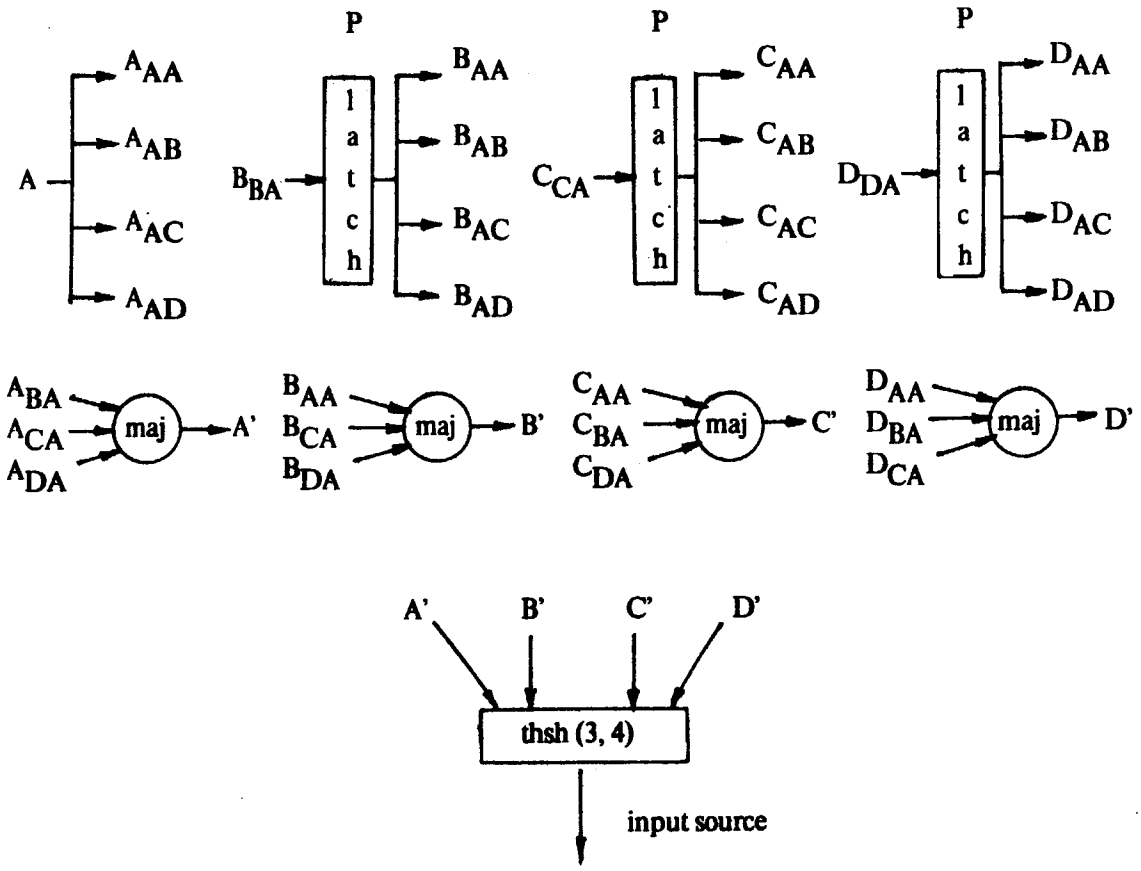
(i) The request source received from *output cntl* is broadcast to all decision boxes. This step is implemented by broadcasting the signal received on A to all decision boxes, as shown in the upper left hand corner of Fig. 5.8b.

(ii) Request sources from other decision boxes are stored in latches.

(iii) Request sources stored in latches are broadcast to all decision boxes.



(a) Decision Box.



(b) Decision box data path for synchronizing merge module A.

Fig. 5.8. An implementation of the decision algorithm.

(iv) The request source for each synchronizing merge module is determined using a majority voter.

(v) Request sources determined in Step (iv) are fed to a threshold circuit *thsh* (3, 4) (See Section 5.1) to derive a new input source.

In each round of joint decision, latches in a decision box should only be activated after new request sources are available from all fault-free synchronizing merge modules in the same redundant module. And then only after the outputs from all these latches have settled down, and a new input source derived from them using majority voters and the threshold circuit, should a decision box return an acknowledgment to its *output cntl* module. Our approach to synchronize the operations in a decision box with those in other decision boxes is to make use of two derived upper bounds δ_1 and δ_2, such that

(1) If a decision box receives a new request source from its *output cntl* module at t, then it will receive a new request source from every fault-free synchronizing merge module in the same redundant merge module at $t + \delta_1$.

(2) If new request sources are available for storing in latches in a decision box at $t + \delta_1$, then a new input source will be available at the output of the threshold circuit *thsh* (3, 4) in that decision box at $t + \delta_1 + \delta_2$.

Based on these two upper bounds, our synchronization strategy is to generate a pulse in a decision box δ_1 seconds after it has received a new *request source* from its decision box, to store new request sources into latches, and then return an acknowledgment to *output cntl* δ_2 seconds later (Fig. 5.8a). We next consider factors that determine δ_1 and δ_2. δ_1 and δ_2 can be computed for an actual implementation based on these considerations.

In the decision box of synchronizing merge module A (Fig. 5.8b), for example, δ_1 must be sufficiently long such that request sources are delivered at signal lines labeled "A", "B_{BA}", "C_{CA}" and "D_{DA}" within δ_1 of each other. We can identify two components for δ_1. One is an upper bound on the variations in propagation delay along paths leading from interfaces between decision boxes and *output cntl* modules to latches in decision boxes. This bound can be derived by taking delay measurements in an implementation. The second component is an upper bound on the difference in arrival time among request sources to be submitted for the same decision, as they are delivered to decision boxes in the same redundant merge module. To calculate this component we must consider the various situations that may occur in other decision boxes as a decision box receives a new request source.

If the new request source received by a decision box is one that has been rejected in a previous decision, a request from the same input port must be pending at the merge module of every synchronizing merge module which has not yet received this request. If the new request source is derived from a packet which has just arrived at the synchronizing merge module containing the decision box, the same packet will arrive at other synchronizing merge modules within a fixed time interval, due to in-phase operation. Thus under either of these two situations, a new request source will arrive at all other decision boxes within a fixed interval, *unless conflict arises at a merge module due to the simultaneous arrival of two output requests.* When a conflict arises, the merge module may be driven into its metastable state and may stay in that state for an arbitrarily long time. The probability that a conflict arises at time t and remains unresolved at time $t + \sigma$ decreases rapidly with σ, for many implementation technologies. We can in practice pick a value α_1 and assume that any conflict that arises at time t is resolved by time $t + \alpha_1$. Under this assumption, then, we can also calculate an upper bound on the difference in arrival time among request sources for the same decision as they are delivered to decision boxes in the same redundant merge module.

To compute δ_2, let us consider the various situations the other decision boxes may be in when a decision box generates a pulse to store new request sources into its latches. Since not all new request sources arrive at these decision boxes at the same time, and each decision box generates a latching pulse δ_1 seconds after it has received a new request source, latches in different decision boxes may not be activated together. After a decision box has generated its latching pulse, it must wait long enough until outputs from all latches in fault-free decision boxes have settled down. Thus if new request sources can arrive at decision boxes as far apart as β then δ_2 must exceed β. Furthermore δ_2 must be long enough to take into account the propagation delay along paths between decision boxes, and the time it takes for the majority voters and threshold circuits to generate a new input source after the latches have stabilized. There is yet another component which must be added to δ_2, to take into account signals generated by failed synchronizing merge modules driving receiving latches into their metastable states. We assume that a latch will always have come out of its metastable state α_2 seconds after it has entered this state, and use α_2 in computing δ_2.

As in the decoding section designs presented in Section 5.2, we deal with stuck-at faults, random pulse trains and random wave trains all at once by storing intermodule messages in latches and assuming that if any of these latches is driven into a metastable state by a faulty signal, it will always come out of that state after α_2 seconds.

There are two metastable state phenomena we have faced in the synchronizing merge module design: two output requests arriving simultaneously at a merge module, and a random pulse train or a random wave train being delivered to a latch. The former occurs during fault-free operation while the latter is strictly a hardware failure symptom. In both cases we have assumed that if a hardware element enters its metastable state at t, then it would have come out of that state by $t + \alpha$, for some fixed α. By using larger and larger α's, this assumption can be made more and more accurate, at the expense of performance. This tradeoff between performance and reliability seems unavoidable in

non-determinate systems.

We next analyze performance compatibility among copies of synchronizing merge modules implemented using the above techniques. There are three factors contributing to phase difference among packets or acknowledgments in the same batch generated by synchronizing merge modules in the same redundant merge module:

(i) the phase difference among packets and acknowledgments in the same batch delivered to the synchronizing merge modules,

(ii) the time it takes a merge module to resolve conflicts. This is necessary because conflicts may arise at some, but not necessarily all, synchronizing merge modules.

(iii) the two different courses of action that can be taken by *output cntl* after the input source is determined.

We have assumed that acceptable reliability and fault coverage can be achieved by upper bounding (ii) with α_1. Under this assumption the performance incompatibility between two synchronizing merge modules is bounded by some parameter which depends only on propagation and gate delays through various paths in and between these modules and in-phase operation can be maintained.

5.4 Design Examples

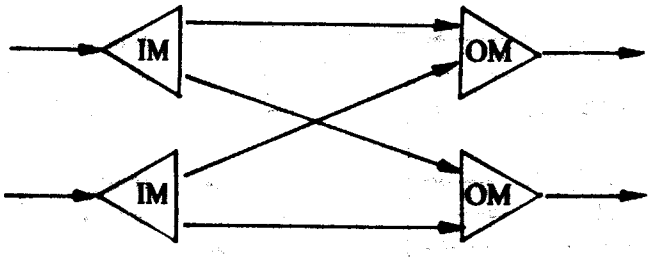
In this section our techniques for maintaining timing synchronization and consistency in redundant packet communication systems are illustrated by applying them to the design of fault-tolerant *routers*.

A router is a 2 input/2 output building block module used in constructing packet networks for the packet communication computer architecture presented in the next chapter. This module receives packets at its two input ports and delivers each received packet at one of two output ports

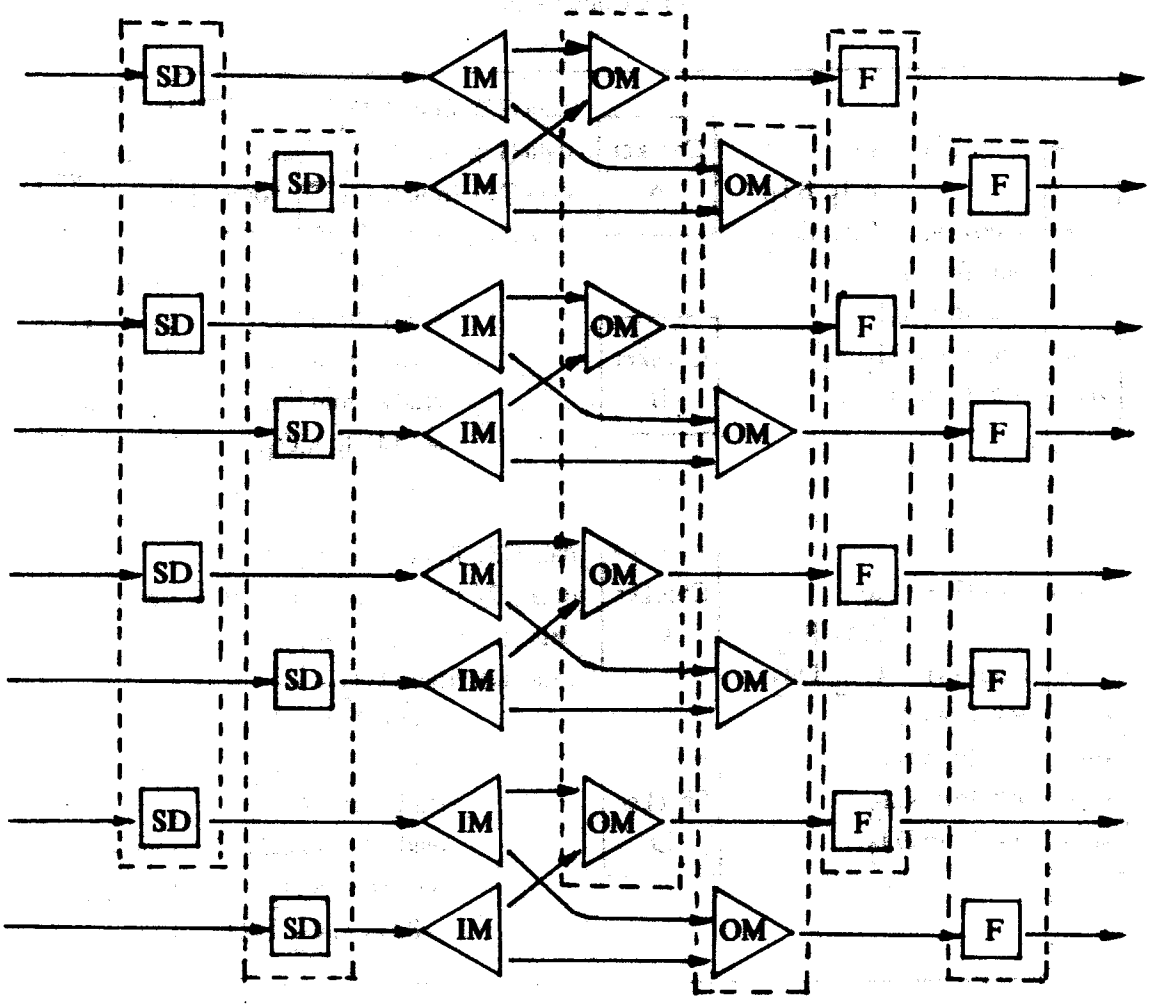
according to a *destination address* carried by the packet, and is designed so that packets to be forwarded at different ports can be processed concurrently. Such concurrency is naturally supported by decomposing the router into two input modules (IM) and two output modules (OM) (Fig. 5.9a). An input module is a sequential machine which examines the destination address in each packet and forwards that packet to the output module specified. An output module is simply a synchronizing merge module. Straightforward application of the redundancy techniques presented in this and previous chapters leads to the redundant router design shown in Fig. 5.9b. Control modules belonging to the same synchronization set and OM modules which must be kept consistent are enclosed by dotted lines. This redundancy scheme can tolerate hardware failures confined to a single unit consisting of two synchronizing decoders, a router module and two fanout modules.

In an implementation of a non-redundant router using off-the-shelf components, it is more practical to implement packet communication using a sequence of 8-bit nibbles[1] delivered under a data link/control link protocol (Fig. 1.1). Under this protocol redundancy can be applied to the data nibble and control link independently. For fault masking the data nibble can either be replicated or encoded. The control link can be protected by replication using techniques presented in previous chapters. A redundancy scheme based on a single-error-correcting Hamming code [29] code is shown in Fig. 5.10. This code uses a total of 12 binary signals to transmit 8 information bits. Each input nibble is decoded by four router units. The data outputs of these router units are voted upon to derive the 12 output bits at an output port. Control signals are quadruplicated and undergo in-phase synchronization as they arrive at the router units. These units also exchange messages to select input sources. For clarity the interunit synchronization paths are omitted in Fig. 5.10. In using a separate control link to synchronize packet communication the delays incurred in different portions of the

1. The term *byte* is commonly used instead of *nibbles* in this context. We prefer the term *nibbles* since we have used the term byte to denote bit fields in a word generated by a redundant module.

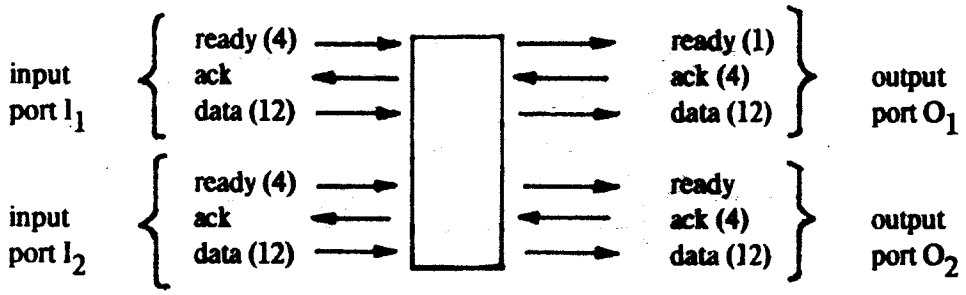


(a) A 2 X 2 Router.

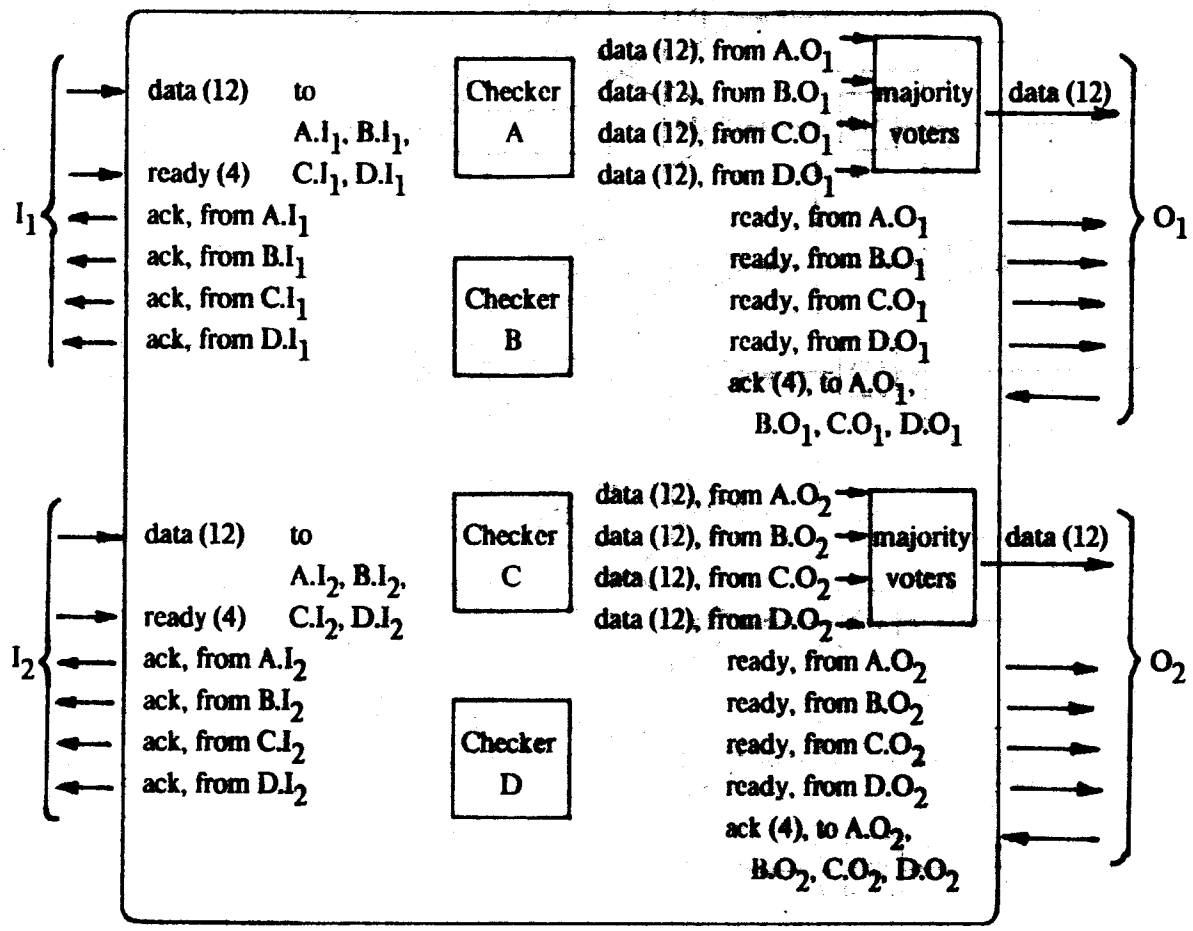


(b) Fault-tolerant Configuration

Fig. 5.9. A fault-tolerant router design based on replicated redundancy.



(a) A Router Unit



(b) The Redundant Router

Fig. 5.10. Hardware structure of a 2 × 2 router using an error-correcting code.

data path must be known. Voter delays, for example, must be taken into account in asserting ready signals. In addition to hardware failures confined to a single router unit, single voter failures in this redundant router can also be tolerated.

Compared with a replication scheme, a coding scheme requires less input and output connections among routers at the expense of more hardware packages and connections within a router. This is advantageous if a more expensive and/or less reliable technology is used for interrouter connection. In the fault-tolerant architecture described in Chapter 6, a redundant router designed to detect hardware failures is used to construct packet networks. This router uses a parity check to protect the data link and has the same hardware structure as that shown in Fig. 5.10. Its design is discussed in more detail in Section 6.3.

5.5 Discussion

In this chapter we have studied hardware implementation of the redundancy management algorithms presented in Chapter 3, and analyzed the fault tolerance capabilities of these implementations under the stuck-at fault model, the random pulse train fault model and the random wave train fault model.

For the synchronizer implementation, different techniques are given for tolerating hardware failures under the three different fault models explained in Section 4.2. Since these three fault models form a strict hierarchy in terms of modeling power, any technique for tolerating hardware failures modeled by one of them is also adequate for tolerating hardware failures modeled by the less general ones.

In the implementation of decoding sections, our approach is to first generate a synchronizer signal that is free of output hazards and runt pulses. Using this signal as a timing signal to store input

signal values into latches, we have shown how to reduce the fault tolerance problem of handling random pulse trains and random wave trains to a timing problem caused by metastable state phenomena in bistable devices. This same technique is also used in synchronizing merge modules to deal with random pulse train and random wave train inputs. The timing problem can be dealt with, if only probabilistically, since in most implementation technologies the probability that a latch remains in its metastable state at time $t + \rho$ after entering that state at t decreases with ρ. It is thus possible to tradeoff reliability with performance in these situations.

We have developed an approach to mask and detect hardware failures in a packet communication system based on maintaining timing synchronization and consistency throughout the entire system. A methodological issue that arises is whether these two properties are necessary for tolerating hardware failures in packet communication systems. It seems that consistency must always be maintained in a byte-sliced module, otherwise a packet batch generated by fault-free slices can have arbitrary contents, and byte slice failures can lead to undetected and uncorrected errors. We next describe a technique for detecting hardware failures under the random pulse train fault model without maintaining timing synchronization.

A popular scheme for fault detection in synchronous hardware systems is to operate two identical hardware units concurrently and detect failures in either one of them by comparing their outputs. Note that there is not enough redundancy in a duplex system for maintaining timing synchronization or consistency using our techniques. Duplication can nonetheless be exploited to detect failures in determinate systems, if hardware failures can be modeled as random pulse trains and, as stated in Section 4.2, C-elements can be used to filter out random pulse trains. A decoder for duplex systems can be constructed using the technique illustrated in Fig. 5.6, except that a C-element instead of a synchronizer is used to activate input latches (Fig. 5.11). A fanout module consists of simply a C-element and a byte detector for deriving the filtering signal from the packet bundle.

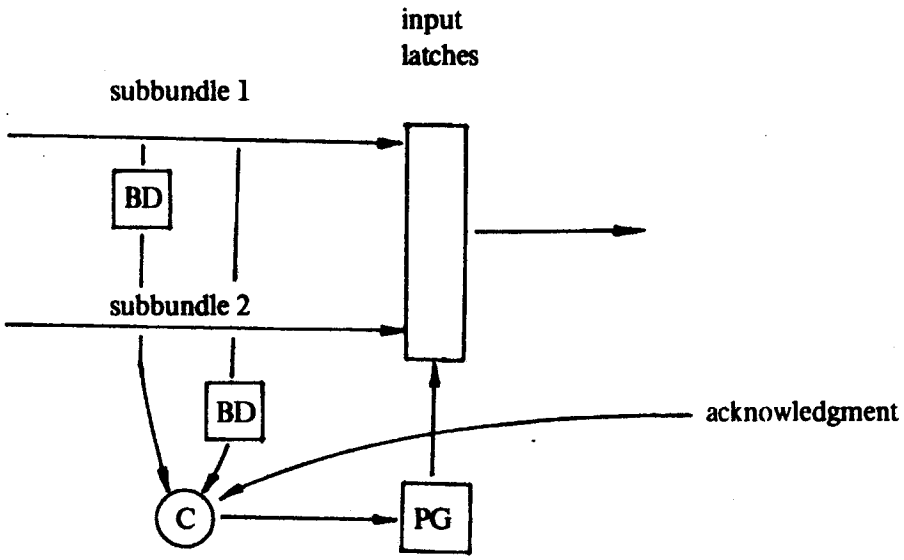


Fig. 5.11. A synchronizing decoder for duplex systems.

This technique can be generalized to detect hardware failures under the random pulse train fault model using other encoding schemes. In such a scheme, a decoder will always wait for one packet from each neighboring fanout module before decoding, while a fanout module will always wait for one acknowledgment from each neighboring decoder before returning an acknowledgment to its p-module.

A similar strategy for *masking* hardware failures is for a decoder to wait until it has received packets from sufficiently many neighboring fanout modules to derive a new error-free input word, and then forward this input word to its p-module. A fanout module is designed to forward a new byte it has received from its p-module to each decoder which has acknowledged the previous byte.

In a redundant configuration based on byte slicing, it can be shown (See Davies and Wakerly [17] for an illustration of this phenomenon) that byte slice failures in redundant modules can cause

fault-free slices in their neighbors to go out of phase by an arbitrary amount. Thus a byte slice that has fallen behind others in its module may not have failed. To maintain communication under the adopted handshake protocol, a decoder should not return acknowledgments to a neighboring slice that has fallen behind before receiving new input bytes from this slice. The decoder should furthermore retain enough information, such as a byte count, to be able to deduce when this neighboring slice has caught up with other slices in its module. Since a neighboring slice can fall arbitrarily far behind, there is no upper bound on the amount of information, and hence the storage for it, that must be kept in a decoder. This decoding strategy is thus theoretically not realizable, although a practical decoder implementation can be constructed using sufficiently large counters.

There is a similar theoretical problem in implementing this strategy in a fanout module. A packet must be retained in a fanout module until it has been delivered to every neighboring decoder, since the contents of an output packet generated by a p-module may depend on the entire stream of input packets it has previously received and processed. A neighboring decoder can again fall arbitrarily far behind and there is no upper bound on the amount of buffering required to store undelivered packets in a fanout module. The question of how much buffering to include in each fanout module, and buffer management techniques, present more of a practical problem. Techniques for estimating the buffering requirements accurately must also be developed since the effectiveness of this strategy in combating hardware failures depends on the accuracy of such an estimation.

By maintaining timing synchronization in a redundant packet communication system, all fault-free byte slices in a redundant module are always in-phase, and the buffering problems that arise in the above scheme is avoided. Instead of buffering facilities, additional hardware and interconnection paths are invested to implement the synchronization algorithm for maintaining timing synchronization.

6. Design of a Fault-Tolerant Packet Communication Computer Architecture

In this chapter we describe a conceptual design for a fault-tolerant data flow processor intended for physics simulation applications. The salient features of this design have been introduced in Chapter 1:

-- High performance and fault tolerance are achieved by using pools of identical hardware units.
-- Communication between processing elements and functional units is supported by packet networks.
-- Hardware in the data flow processor is organized by a packet communication architecture.

As observed in Chapter 1, there are no stringent reliability requirements inherent in physics simulation applications. We are interested in adding fault tolerance capabilities to the non-redundant system primarily to improve maintainability and availability. System throughput is improved if hardware failures can be masked, especially since it is not unusual for a numerical computation in physics simulation to execute for many hours.

Fault-tolerant mechanisms employed in the redundant data flow processor are organized according to a *dynamic redundancy scheme*. In this scheme hardware failures are detected and diagnosed, the data flow processor is repaired, and then afflicted subcomputations are reexecuted on the repaired, possibly degraded, system. We assume that programs for the data flow processor are prepared on a host machine, loaded into the data flow processor, and then executed. This host machine is also assigned the tasks of configuration control, and of coordinating diagnosis, repair and recovery activities with program execution. We will explain the strategy to be implemented on this host machine, and the application of hardware redundancy techniques developed in previous chapters and some packet encoding techniques to implementing hardware modules in support of this strategy. A fault-tolerant implementation of this strategy on the host machine can be constructed

using conventional techniques, and will not be considered.

The hardware organization of the data flow processor and its operation are explained in Section 6.1, where we will focus on those aspects of its operation relevant to fault tolerance considerations. The dynamic redundancy scheme is explained in Section 6.2. Hardware module designs to satisfy fault tolerance requirements imposed on them by the dynamic redundancy scheme are presented in Section 6.3. Strategies for incorporating additional hardware into a packet network to support rapid repair are discussed in Section 6.4.

6.1 A Packet Communication Computer Architecture

6.1.1 Hardware Organization

Hardware in a packet communication computer architecture is organized as an interconnection of self-timed modules which communicate by sending packets to each other. Each packet is transmitted as a sequence of nibbles between modules. Packet nibbles are delivered and received by hardware modules using the ready/acknowledge protocol depicted in Fig. 1.1. Each *module port* consists of a bundle of data wires and a pair of control wires (Fig. 1.1a). Packet communication is synchronized by sending control signals over the control wires. Availability of a new nibble at a connection is signaled by sending a *ready* signal over the ready wire, its receipt by returning an *acknowledge* signal over the acknowledge wire. *Ready* and *acknowledge* signals are represented by signal transitions (Fig. 1.1b) on the respective wires.

The major modules in the data flow processor (Fig. 6.1) are processing elements (PEs), specialized functional units (SFUs), a routing network and an allocation network. Scalar operations are processed in the PEs and SFUs. The networks support packet traffic among the PEs and SFUs.

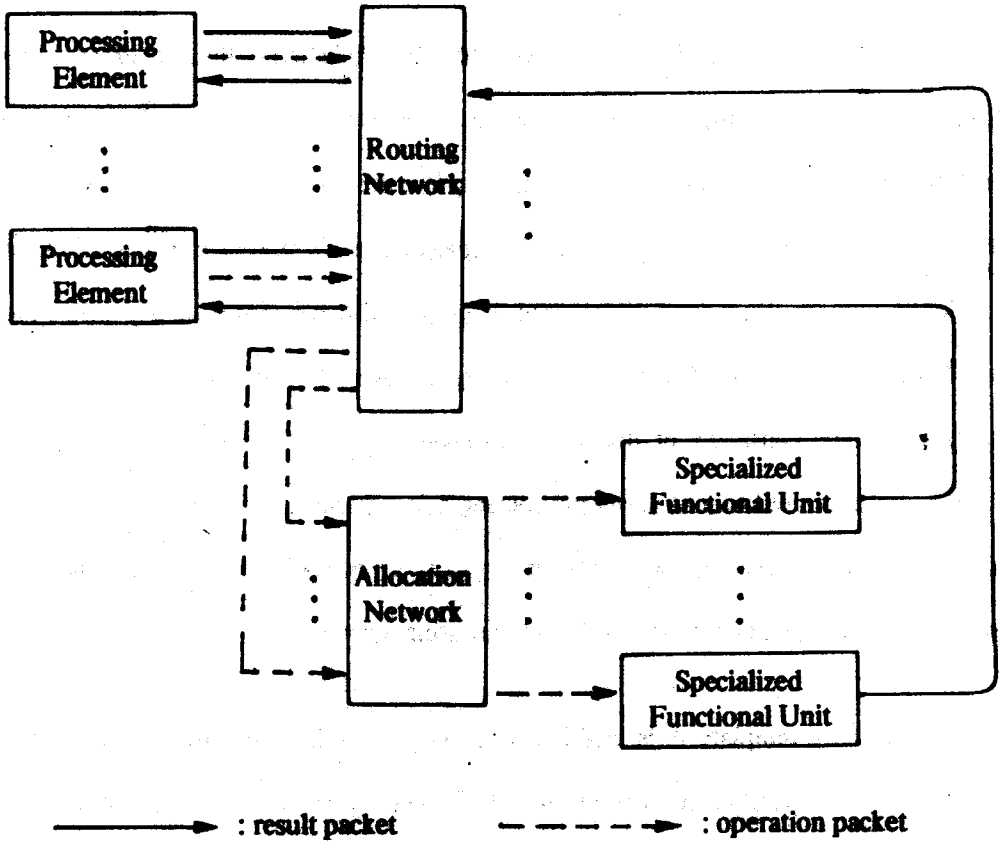


Fig. 6.1. Hardware architecture of the data flow processor.

Operating Principles

A machine level program for the data flow processor is stored in the PEs as a set of *activity templates* [20]. An activity template A contains an *operation code* and the *addresses* of one or more activity templates, called A's *target templates*, which should receive the result generated by processing A. An activity template is uniquely identified by its address, which has two components: a destination tag which specifies the PE in which the template resides, and the location of the template within that PE.

Operations are divided into two classes, according to whether they are executed by a PE or by a SFU. When activity template A is enabled (See below for a discussion of the enabling conditions) and the operation it specifies is executable on a PE, the operation is applied to the operands A has received and the result of the application is dispatched to A's target templates. If a target template B resides on another PE, a copy of the result is tagged with B's address and a nibble count to form a *result packet*, and delivered to B via the routing network. For an operation executed on a SFU, an *operation packet* consisting of the operation code, operands and the target template addresses is formed and delivered to a SFU via the routing network and the allocation network. At a SFU the operation specified in an operation packet is applied to the operands and result packets are generated and dispatched to the target templates via the routing network.

An activity template A is enabled when two conditions are met. First of all the operands required for the operation must have arrived. The second condition is a consequence of organizing machine level programs in the data flow processor to support pipelining and iteration. Suppose that in such a program activity template A sends results to activity template B residing in another PE. To avoid deadlock [38], an operand sent from A to B must be processed before A can send B a second operand. The execution of A and B are synchronized by conditioning each activation of A on receiving an *acknowledge packet* from B. This acknowledge packet is transmitted when B is processed with the previous operand received from A. Thus before an activity template can be executed, it must have received the necessary acknowledgments from its target templates. A detailed explanation of this synchronization scheme is given in [26]. Under this scheme every result packet transmitted through the networks is acknowledged by an acknowledge packet returned by the target template. We will also make the assumption that *each operation packet contains exactly one target template address*. This assumption results in no loss of generality since the result obtained by processing an operation packet can be further distributed through its target template. In fault-free

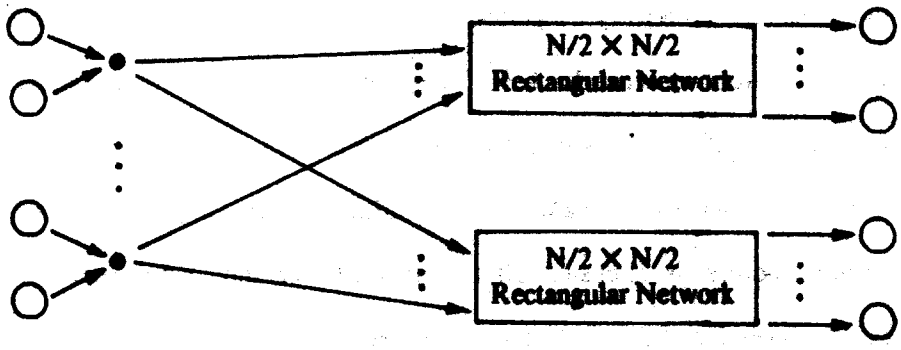
operation under the above synchronization scheme and this assumption on operation packets, a PE will receive exactly one acknowledge packet for every result packet or operation packet it delivers to the routing network. This property will be used to coordinate recovery activities with normal program execution in the dynamic redundancy scheme.

A PE provides storage for activity templates, executes simple operations, sends packets to and receives packets from the routing network. A SFU is designed to execute complex operations, such as floating point arithmetic, efficiently, with additional capabilities to receive operation packets from the allocation network and send result packets to the routing network. Implementation techniques for these hardware modules as discussed after their fault tolerance requirements have been determined. In the remainder of this section we take a closer look at the structure of the routing network and the allocation network.

6.1.2 Packet Networks

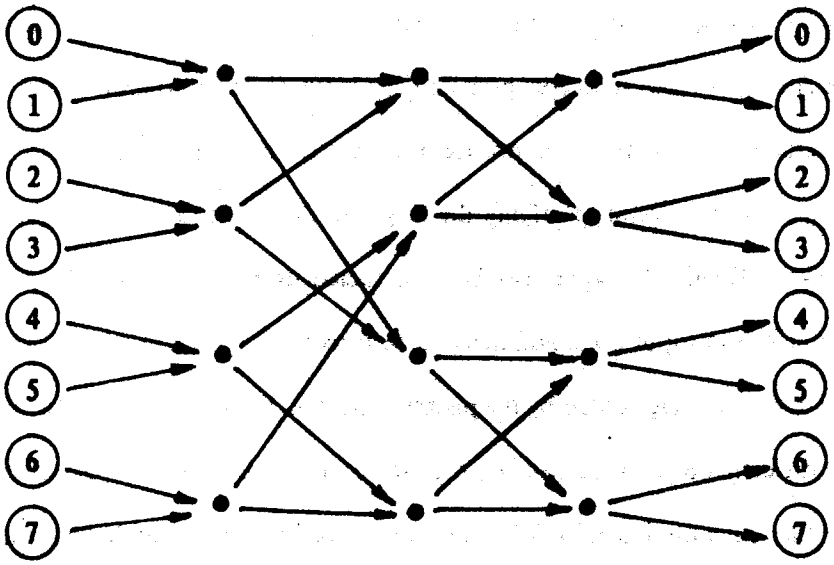
An N × N routing network, with N input ports and N output ports, supports packet communication among N PEs. It accepts packets at its input ports and transmits each packet at the output port specified by a bit field comprising a header or destination tag of the packet. The destination tag and a nibble count specifying the length of the packet are contained in the first few nibbles of a packet. Routing networks can be constructed using 2 × 2 routers. A 2 × 2 router receives packets at its two input ports and delivers each received packet at one of two output ports according to the destination tag carried by the packet. The 2 × 2 router is designed so that packets to be forwarded at different output ports can be processed concurrently. We have described a hardware organization for the router module (Fig. 5.10a) and several designs for fault-tolerant routers (Fig. 5.10 and Fig. 5.11) in Section 5.4 to illustrate our hardware redundancy techniques.

Methods and techniques for tolerating hardware failures in routing networks will be illustrated using *rectangular* routing networks. An N × N *rectangular* network is built from 2 × 2 *routers* by the recursive construction illustrated in Fig. 6.2. An N × N network so constructed has $\log_2 N$ stages



: 2 × 2 router

(a) Recursive construction.



(b) An 8 × 8 rectangular network.

Fig. 6.2. Rectangular routing networks.

each of which contains N/2 routers. All packets sent to an output port of the routing network, regardless of their sources, have identical destination tags. Routers in succeeding stages in the network examine successive bits in a destination tag to forward the packet along the proper path. Path control in a routing network is distributed among the routers. There is no centralized control mechanism whose complexity must grow with network size and which may become a performance bottleneck. Many packets can be forwarded concurrently to provide a high throughput rate. One of the design objectives for a fault-tolerant routing network is to retain all these characteristics during fault-free operation: decentralized path control, parallel processing, and asynchronous nibble-serial communication.

In the data flow processor the allocation network receives operation packets from the routing network and distributes them among the SFUs. Each of its input ports has its own routing network address. In the data flow processor we use a rectangular allocation network which also has the topology shown in Fig. 6.2, constructed out of 2 × 2 allocators. An allocator receives operation packets from its two input ports and forwards them at its output ports as these output ports become free. It is possible for operation packets to be temporarily "trapped" in a section of the Allocation Network waiting for service even though SFUs not reachable from this subnetwork are free. Such trapping has the pleasing property of automatically diverting other operation packets from the congested subnetwork. We next present an allocator implementation constructed from determinate modules and merge modules. This implementation can be rendered fault-tolerant using exactly the same techniques illustrated in Section 5.4 for router modules.

The implementation strategy we have adopted for the allocator is to generate *service requests*, for new input packets, and *availability tokens*, when output ports become free. Service requests from the two input ports are then merged together, as are availability tokens from the two output ports. Each service request is matched with an availability token, and a packet is transferred from the

requesting input port to the available output port. In the hardware implementation (Fig. 6.3), service requests are generated by *input port controllers*, availability tokens by *output port controllers*, and they are matched at a *matcher* module which also handles packet transfer from input port controllers to output port controllers.

6.2 A Fault Tolerance Strategy Based on Dynamic Redundancy

It is possible to apply static redundancy techniques uniformly in the data flow processor to implement a fault masking capability in hardware. We have presented two router designs in Section 5.4 appropriate for constructing fault masking routing networks, and pointed out that similar techniques are applicable to allocation networks. We have also designed a router module, based on parity checks, to support fault detection and, again, similar techniques are applicable to the design of fault-detecting allocator modules. This router design is described in the next section. A packet network constructed with these fault-detecting modules has considerably fewer intermodule connections than one based on replication (using the type of router modules illustrated in Fig. 5.10),

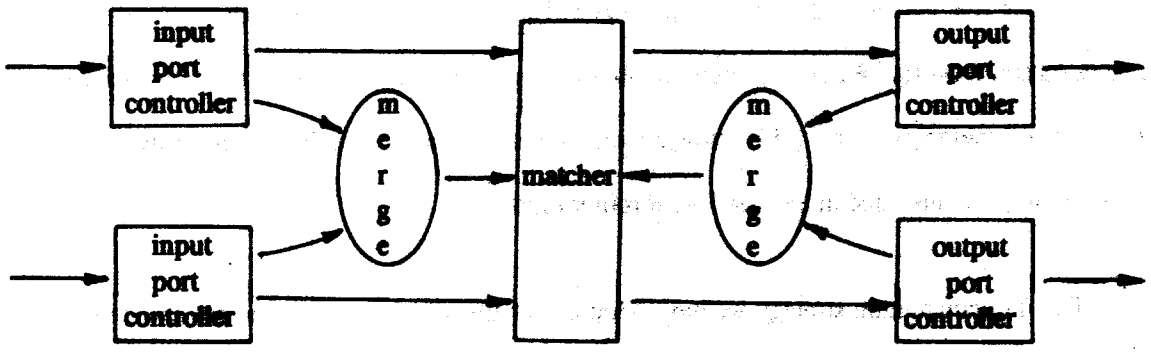


Fig. 6.3. A hardware implementation of an allocator module.

and fewer hardware packages and intermodule connections than one based on error-correcting codes (using the type of router modules illustrated in Fig. 5.11). It is thus attractive to develop a fault masking strategy which can exploit the advantages of network designs based on fault-detecting modules. Such a strategy leads to a *dynamic redundancy* scheme.

In a dynamic redundancy scheme, failures are masked through hardware-implemented fault detection, diagnosis, repair, followed by reexecution of the afflicted subcomputations. To support the dynamic redundancy scheme, each PE must store additional information that is not used in a non-redundant data flow processor. A copy of every result and operation packet delivered to the *packet transport and processing* (PTP) subsystem consisting of the packet networks and the SFUs must be kept until the packet is acknowledged. At each PE the sender of every received packet must be known. This information can be maintained by associating each operand position in an activity template with a template address for returning acknowledgments. The sender of each acknowledgment packet and retransmission request (See below) must also be identifiable.

The dynamic redundancy scheme assumes that the hardware implementation of the data flow processor has the following properties, even in the presence of hardware failures.

(1) Each packet received by the routing network is delivered at the output port specified by its destination tag, either to a PE or the allocation network. Each packet received by the allocation network is delivered to a SFU. Specifically, neither packets nor packet nibbles will be lost in the networks.

(2) Each packet delivered to a network output port is either error-free or flagged as erroneous.

(3) Target template addresses carried in an operation packet are always delivered free of error.

(4) For every operation packet received, a SFU will deliver either an error-free result packet or

one tagged as erroneous to the target template specified.

(5) Every acknowledge packet and retransmission request (See below) is delivered free of error to its destination.

Techniques to design redundant routers, allocators, PEs and SFUs that can be used to implement a data flow processor with these properties, even when up to one hardware package in each module has failed, will be presented in Section 6.3. The fault tolerance strategy is as follows.

Any packet which has encountered a faulty router or allocator in its journey through the networks will be marked as such upon delivery to the destination PE or SFU. When a SFU receives an operation packet tagged as erroneous, it will generate a result packet, tag it as erroneous, and forward it to the target template specified in the operation packet. If a PE receives a result packet tagged as erroneous, it signals the host machine, which in turn signals the other PEs to stop sending packets to the routing network. All packets in transit will arrive at their destinations after a finite time period, which can be determined from hardware parameters specified for the PTP subsystem. After this time period the PTP subsystem can be repaired under the direction of error signals generated by fault detectors in this subsystem. After repair the PEs are restarted. A PE which has received a contaminated packet will issue a *retransmission request* instead of an acknowledgment to the sender. Program execution can otherwise proceed normally.

Activity templates representing a machine level program, and a complete intermediate state of the computation in progress, are stored in the PEs. If only routers, allocators or SFUs have failed, the computation can always be restarted from the intermediate state stored in the PEs and run to completion after the networks have been repaired. If failures occur in the storage components of a PE, it may be necessary to abort the computation in progress, since the intermediate state may become inconsistent. Storage failures thus must be masked to achieve complete fault masking.

Failure in other components of a PE need not be masked in hardware, but the activity templates and partial intermediate state stored in it must be relocated before processing can resume. If an activity template A is relocated, the entire activity template set must be relinked so that other templates having A as a target will contain the new address of A. It thus seems desirable to mask all failures in PEs locally in hardware. Communication between PEs and the host machine to coordinate repair can be implemented with an interprocessor bus. Fault-tolerant busing structures have been presented in [30] and [67].

The PTP subsystem must be repaired after failures are detected. A failed module can be replaced or repaired in place manually. Availability is improved by reconfiguring around the failed module automatically and then repairing the failed module off-line. For a computation in progress to proceed successfully, the full functionality of the routing network must be retained, i.e., packet communication between any input port and any output port must be maintained. Two strategies for incorporating spare routers and data paths into a routing network are discussed in Section 6.4. Under these strategies the full capability of a routing network is retained so long as spares are not exhausted. These strategies are also applicable to allocation networks. A SFU failure can be repaired by simply taking it off-line. An allocator will not forward operation packets to any SFU which has stopped acknowledging inputs. The machine architecture is gracefully degradable with respect to SFU failures in this sense. SFU failures have no effect other than degraded performance across the PTP subsystem boundary.

The dynamic redundancy scheme we have described is built directly on the execution control mechanism in a data flow processor. It has the merit that extensions to the execution control mechanism are incorporated in low level hardware functions and require no extra programming effort to achieve fault tolerance.

6.3 Module Design

Our basic unit for fault tolerance considerations is a *package*, which receives input signals and delivers output signals through its *terminals*. Hardware modules are constructed using packages. We also assume that the mean time to repair is much shorter than the mean time to failure. Modules described in this section are hence designed to tolerate up to one package failure per module. In this section we present hardware redundancy and packet encoding techniques to support the five fault tolerance properties stated in Section 6.2. We first describe the encoding techniques:

-- Control signals are generated in quadruplicate, from four failure-independent packages.

-- Each data nibble is protected by a parity bit. The nine bits of a parity-encoded nibble are generated from nine failure-independent packages.

-- Each packet nibble whose error-free transmission must be guaranteed is expanded into three nibbles. The second and third nibbles in each triplet are obtained by rotating bits in the given nibble one and two positions to the right, respectively:

given nibble:	$b_0b_1b_2b_3b_4b_5b_6b_7b_8$
2nd nibble:	$b_8b_0b_1b_2b_3b_4b_5b_6b_7$
3rd nibble:	$b_7b_8b_0b_1b_2b_3b_4b_5b_6$

This encoding scheme can be regarded as an implementation of triple modular redundancy in time instead of in space. Nibbles in *nibble count* fields and *template address* fields, and identification tags in acknowledgment packets and retransmission requests are protected using this technique.

-- An all 0's data nibble is appended to the tail of each packet. This nibble is used to flag packets

which contain erroneous nibbles. It is set to all 1's by the first module which detects the parity violation, and is otherwise retransmitted as received. The packet is accepted as error-free only if its flag consists of all 0's.

Using the hardware redundancy techniques developed in previous chapters and the above encoding techniques routers, allocators and SFUs can be implemented with the following fault tolerance properties:

-- If at most one of the four control signals delivered on each quadruple is faulty, its pathological effects can be masked.

-- If at most one data wire in each parity-checked data link carries faulty signals, the nibble counts and addressing information in each packet can be retrieved. The last nibble in the corresponding output packet will be flagged, i.e., will not be all 0's, if any packet nibble has violated the parity check.

-- If at most one package in the module has failed, the above capabilities are not impaired and faulty signals are delivered on at most one output control wire and one output data wire at any module port.

We note that it is in fact possible to mask all single package failures in routers, allocators and SFUs with these fault tolerance capabilities if every packet nibble is triplicated using the rotate-and-repeat encoding scheme given above. This approach to fault masking leads to lower performance during fault-free operation, as compared with the dynamic redundancy scheme explained in the last section.

We next illustrate module design techniques with a router module design and a processing element design. These techniques are also applicable to implementing the desired fault tolerance capabilities in allocators and specialized functional units.

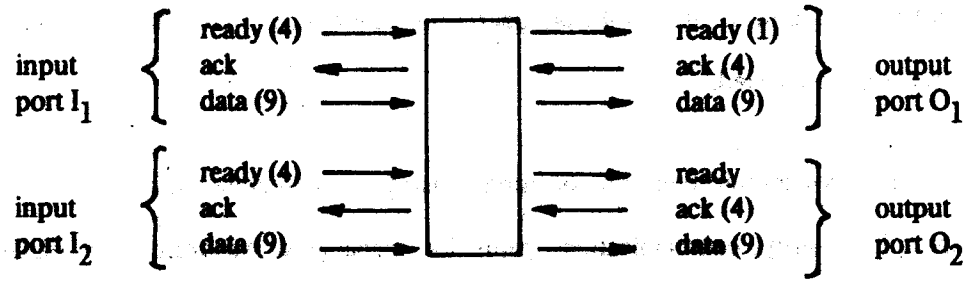
Router Module Design

A *redundant* router module (Fig. 6.4), just as its nonredundant counterpart, receives packets at its two input ports, and delivers each received packet at the output port specified by a destination tag carried in the packet. Packet receipt and forwarding are synchronized by control signals delivered in quadruplicate. Packet nibbles are parity encoded. The redundant router is implemented using four *checker* packages and nine *voter* packages (Fig. 6.4). Each checker (Fig. 6.4a) has two input ports and two output ports. Control signals generated at the corresponding ports of the four checkers are grouped together at each port of the redundant router module. Thus, for example, the acknowledge signals generated by the four checkers at their I_1 input ports (Fig. 6.4b) are grouped together at input port I_1 of the redundant router. Data nibble outputs from the corresponding output ports of the four checkers are collected together and voted upon at the voters to derive outputs for an output port of the redundant router. In Fig. 6.4, the number of wires represented by each arrow is given in its label in parentheses. This number is omitted if the arrow represents a single wire.

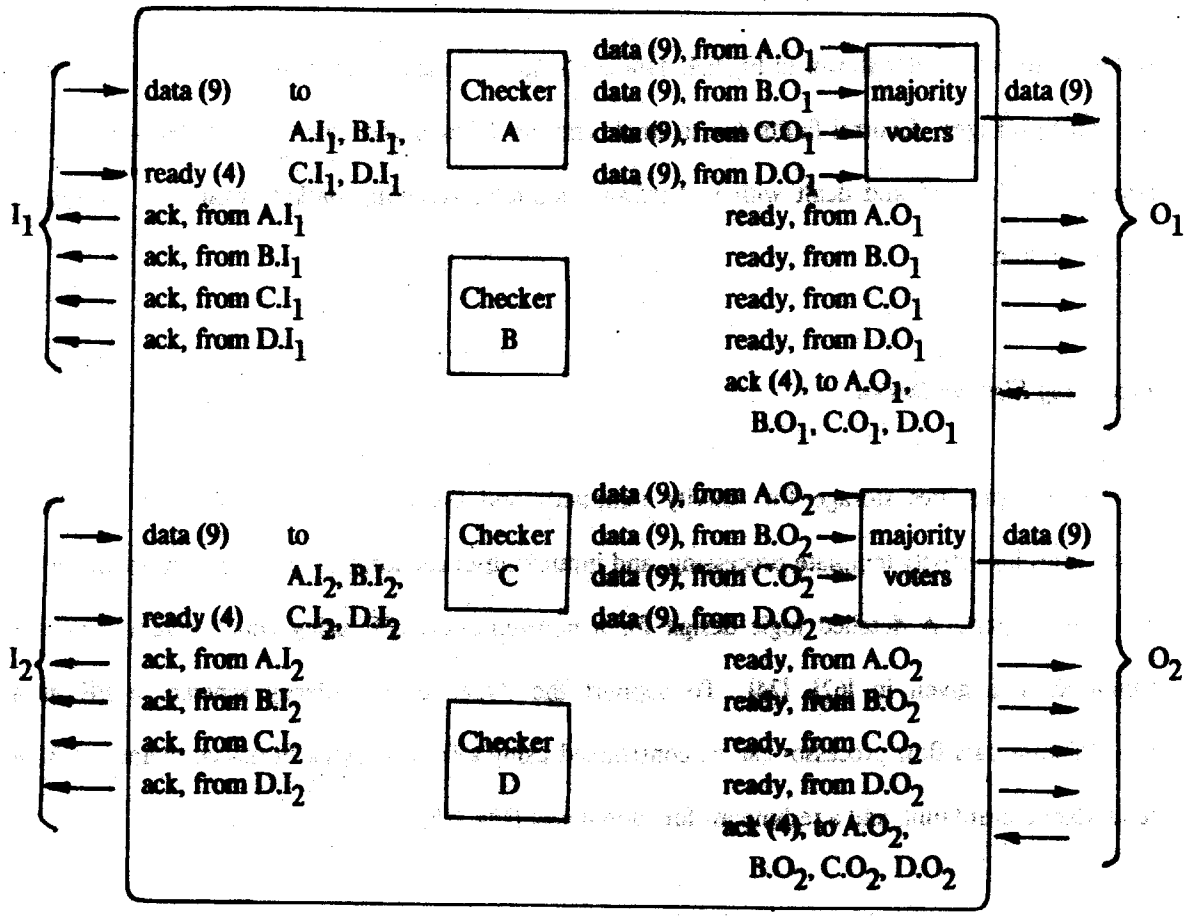
A checker receives all input control and data signals delivered to the router, and implements several fault tolerance capabilities in addition to packet routing:

-- mask control signal failures when at most one control signal in each quadruple is faulty.

-- deduce the error-free nibble for every data nibble encoded in triplicate using the rotate-and-repeat scheme given above.



(a) A Checker Unit



(b) The Redundant Router

Fig. 6.4. Hardware structure of a redundant 2 × 2 router based on parity codes.

-- set the last packet nibble to all 1's if the last packet nibble it has received is not all 0's or if parity violation has been detected.

-- maintain timing synchronization and consistency using the techniques developed in previous chapters. For clarity, communications paths for exchanging synchronization signals and messages among checker packages to implement these redundancy management functions are omitted in Fig. 6.4.

In the redundant router shown in Fig 6.4. data faults due to single checker failures will be masked at the voter packages. Control faults due to single checker failures and data faults due to single voter failures are detected and dealt with in hardware modules receiving packets from this redundant router module.

Processing Element Design

A PE provides storage for activity templates and their operands, as well as functional capabilities for activity template processing, and input/output capabilities for packet communication and error report. A detailed logic design for a non-redundant PE using commercially available components is given in [62], [24]. To support the dynamic redundancy scheme, a PE in a fault-tolerant data flow processor can be constructed using a fault masking bit-sliced memory [13], a redundant control unit, and a redundant functional unit (Fig. 6.5).

A partial state of the computation in execution is stored in the bit-sliced memory. For the computation to be recoverable after single package failures, all such failures must be masked along the data path used to retrieve this information from the memory to the host machine. This is achieved by using a bit-sliced memory protected by an error-correcting code to store this information, and a control unit with the same structure and operating principle, and hence the same fault tolerance

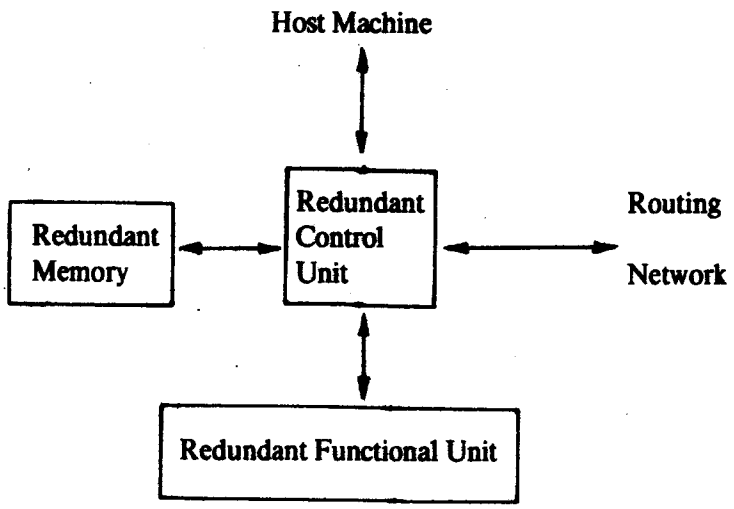


Fig. 6.5. Design of a fault-tolerant processing element.

capabilities, as the redundant router module shown in Fig. 6.4. The control unit consists of four failure-independent packages each of which receives all input signals delivered to the control unit. The outputs of these four packages are grouped together or voted upon to form outputs of the redundant control unit.

Addresses and data transmitted between the redundant control unit and the bit-sliced memory system, and those transmitted between the control unit and the host machine, are encoded using an error-correcting code. Each bit-slice in the memory system stores one bit of a data word and has its own address decoder. Any hardware failure confined to within one bit-slice thus affects at most one bit of a data word and consequent errors can be corrected. Packet nibbles transmitted between the redundant control unit and the routing network are parity-encoded. Since the redundant control unit has the same fault-tolerance capabilities as the redundant router, single package failures in the control unit cannot cause undetected erroneous packets to be delivered to another PE.

The functional unit is the only subunit in a PE that need not be completely fault-tolerant. Package failures in it must nonetheless be detectable. Many redundancy techniques are available for detecting failures in functional units. For a commercially available LSI functional unit chip, it is cost effective to detect single chip failures through duplication and mask these failures through triple modular redundancy, as desired. Communication between the redundant control unit and the redundant functional unit can also be protected using either an error-correcting code or an error-detecting code, depending on whether failures in the functional unit are to be masked or detected.

6.4 Network Repair Strategies

When a network hardware failure is detected in the dynamic redundancy scheme, all PEs will stop sending packets to each other and after a predetermined time period the networks will be dormant. Before normal processing can resume the failed unit must be located and the networks must be repaired. In this section we will illustrate two repair strategies using routing networks.

The first step in any repair procedure is to locate the failed router. Each checker package can generate an error signal upon detecting a parity violation. Failures in the last network stage are detected directly by parity checkers in the PEs and SFUs. Since checkers are quadruplicated in each router, PE and SFU, two or more error signals will be generated for each legitimate complaint under the single package failure assumption. These error signals can be used to locate the failed router. Further diagnosis will be necessary to locate the failed package(s).

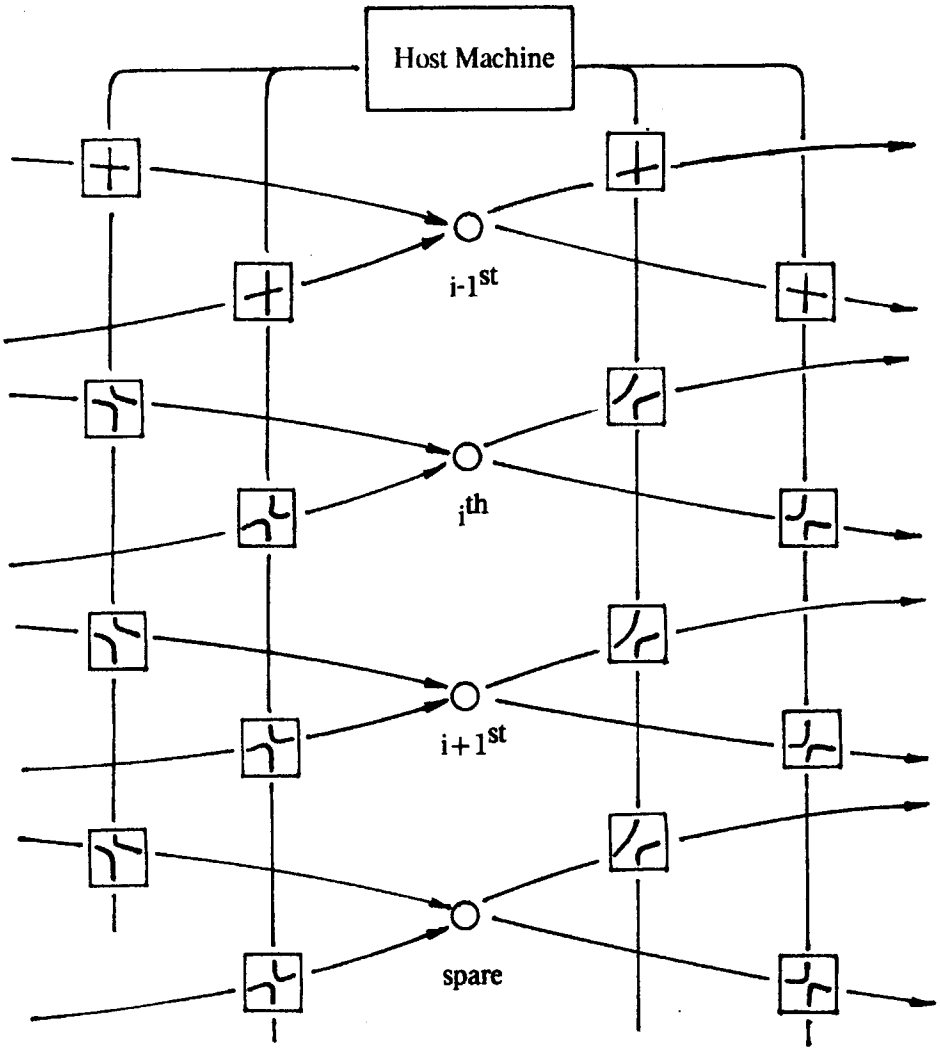
The most straightforward repair procedure is to make use of error signals generated by checker packages to locate the failed unit and then replace it *manually* with a spare. This procedure requires no additional hardware, but system availability is directly related to the availability of maintenance

personnel. The personnel requirement can be reduced by incorporating self-repair features into a network. The error signals will be monitored by the host machine which will direct repair activities. Additional modules or data paths must be incorporated directly into the routing network to support self-repair.

In the self-repair scheme illustrated in Fig. 6.6, a number of spare modules are appended to each routing network stage, switched in electrically to replace failed modules. Switching arrangements are incorporated systematically using *switch* packages, which have been introduced in [36], to support system reconfiguration. A switch can be set in one of two modes, either "crossing" or "bending" (Fig. 6.6a) the pair of input leads to the pair of output leads. Spare routers are interspersed with active routers. The reconfiguration capability of this switching arrangement is illustrated in Fig. 6.6b where the i^{th} router is assumed faulty. Note that in this scheme a spare router cannot replace any faulty router below it in the column. Control signals for setting the switches can also be carried in the interswitch connections. This repair scheme requires many additional data paths and packages, and must be further enhanced to tolerate switch failures. It is thus practical only when the additional hardware costs are acceptable and the switches are much more reliable than the router modules. One technique to tolerate switch failures is to connect each switch to more than one neighboring switch so that an immediate neighbor which has failed can be bypassed. These switches are called *ripplers* in [58].

The additional data paths introduced can also be used for off-line diagnosis, testing out the routers systematically with pregenerated test patterns. In the configuration shown in Fig. 6.6, the i^{th} router can be tested by the host machine while the remaining routers carry the packet traffic. The fault detection mechanism in the dynamic redundancy scheme assumed single package failures in each router. Multiple package failures or lurking failures which have not yet manifested themselves are not detected. Network reliability can be further improved by testing the routers for these failures

(a) The *switch* module



(b) Reconfigurable network stage

Fig. 6.6. A reconfiguration scheme for self-repair.

periodically or after detecting a fault in software.

In the above strategy the topological and operation characteristics of a rectangular routing network are retained after reconfiguration. In a rectangular network any router, except for those in the last stage, can be paired together with a neighbor in the same stage such that the two can be used interchangeably in packet routing. If one router in a pair fails, its duty can be taken over by its partner and the network can continue to operate, possibly with degraded performance. This scheme can be implemented by adding two input ports and two output ports to each redundant router. If there is a path in the nonredundant network from router A to router B, a new path between A and B's partner is added. The redundant paths incorporated into an 8 × 8 network (Fig. 6.2b) are shown in Fig. 6.7. The last stage can be repaired by using the previous scheme. A packet can be forwarded to its destination along two different paths at each enhanced router. Both of these paths can be used during normal operation when all routers are fault-free, or one of them may be designated a spare to

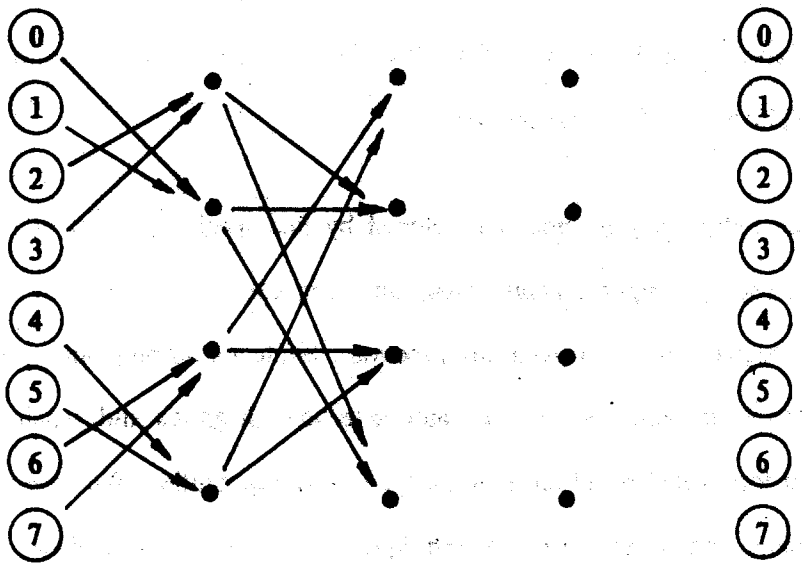


Fig. 6.7. A reconfigurable 8 × 8 network with redundant paths.

be used only when the other path is blocked by a failed router. Information on the location of failed routers can be distributed by the host machine during repair, to disable connections to failed routers.

The host machine can keep a count of the number of failures reported for each router and take it off-line only when a predetermined maximum failure rate is exceeded. Spare modules can then be better utilized when transient failures dominate. We also note that neither of these repair schemes require recomputing destination addresses in a partially executed data flow program to complete its execution after being interrupted by a network failure.

6.5 Discussion

The STAR computer [8] and the FTSC computer [52] are two examples of fault-tolerant computing systems based on dynamic redundancy. Both of them have a bus-oriented architecture designed for executing sequential programs. In this chapter we have presented a dynamic redundancy scheme for masking hardware failures in a multiprocessor system designed to execute parallel programs organized by data flow concepts. These programming concepts and the quest for high performance also distinguish our work from other fault-tolerant multiprocessor projects such as the FTMP system [30] and the SIFT system [67].

The dynamic redundancy scheme is developed by first specifying a set of fault tolerance capabilities for the *packet transport and processing* subsystem, consisting of the packet networks and the specialized functional units, in the data flow processor. Strategies for program execution, and for coordinating fault-related activities such as fault detection, diagnosis and repair with normal execution are then formulated based on these fault tolerance capabilities. We have also explained hardware redundancy and packet encoding techniques for implementing hardware modules and subsystems to support these strategies. Redundant hardware is incorporated and operated according to the redundancy management methodology developed in previous chapters.

The packet transport and processing subsystem is designed to be constructed out of hardware modules, instead of with busing structures or communications technology. The fault tolerance capabilities specified for this subsystem have been chosen primarily because it is quite straightforward to both develop a system strategy based on these capabilities, and to implement them using familiar hardware redundancy concepts at the module level. The fault tolerance and maintainability features of the resulting system design can be precisely characterized and, if only informally, verified. A different set of fault tolerance capabilities for the packet transport and processing subsystem can be adopted for system design, leading to alternative system and implementation strategies. When detailed logic designs and hardware failure rates are available for a hardware implementation, alternative schemes should be carefully evaluated with reliability models [41] to determine their cost-effectiveness.

We have assumed that hardware packages fail under normal use, and that failures are readily repaired. The redundancy schemes have thus been presented assuming at most one package failure in each hardware module. As long as there is at most one failed package in any module, the computation in progress can always be completed. If the processing elements are not designed to mask single package failures in their functional units, it may be necessary to relocate the activity templates and the partial state of the computation stored in a failed processing element, and relink the activity templates, before program execution can be resumed. The redundancy techniques can be extended to accommodate multiple package failures by using more packages in each router and more elaborate coding techniques. In a physical realization several packages can share a physical unit as long as the physical system is partitioned so that under the most common failure modes at most one package in each module can fail.

We have demonstrated how to methodically deal with hardware failures in a practical implementation of a highly parallel data flow processor, with no impact on its programmability. We

have explained how hardware failures can be masked when the architecture is programmed in a restricted data flow language. Another operational restriction is that every packet transmitted over the packet transport and processing subsystem is acknowledged by another packet. The fault tolerance techniques explained in this chapter are directly applicable whenever the hardware architecture is programmed under these restrictions. It is expected that more sophisticated system strategies must be developed to incorporate fault tolerance cost-effectively into more advanced data flow architectures [1], [66].

7. Conclusion

In this chapter we present a summary of results and suggestions for further research.

7.1 Summary of Results

We set out to answer the question:

"How can hardware failures be tolerated in a self-timed hardware system organized by a packet communication architecture and designed to execute data flow programs ?"

Our answer is provided in two parts. In the first part we study the general question of designing redundant packet systems for fault tolerance, and examine the issues of redundancy organization, redundancy management and fault modeling. In the second part we study architectural issues in the design of fault-tolerant data flow processors. Our results in these two areas are presented next, followed by an evaluation of whether these results have provided a satisfactory answer to the above question.

7.1.1 Fault Tolerance in Self-Timed Hardware Systems

Research reported in this thesis has been conducted as part of an effort to develop a design methodology for constructing computer systems with suitable performance, programmability, fault tolerance and modularity characteristics. In particular, we have studied the problem of achieving fault tolerance in self-timed systems organized by a packet communication architecture. In the past there are significant speed and economic penalties for constructing self-timed systems using off-the-shelf components. These disadvantages are greatly alleviated in custom LSI chip design, an implementation approach appropriate for constructing large parallel processing systems for high speed numerical computation.

Many stuck-at faults cause hangups in self-timed hardware systems whose modules interact via asynchronous handshake protocols. This property is often cited as evidence that self-timed systems provide natural support for fault isolation and fault diagnosis. There exists no methodology, however, for incorporating redundant hardware into self-timed systems for fault tolerance, prior to our work.

Our major result in this area is a complete methodology for incorporating redundant hardware into a class of packet systems for fault tolerance. This class includes all determinate systems and non-determinate systems constructed with merge modules. In the course of this investigation, we have addressed the following issues:

Structure of Redundant Modules

We have presented a byte-sliced hardware organization for redundant packet communication modules. Byte slices in a redundant module can be designed and constructed from functional specifications for the nonredundant module and the chosen encoding scheme. We have also described a class of asynchronous packet communication protocols for use in these modules.

Fault Modeling

Hardware failures in redundant modules are characterized by a stuck-at fault model, a random pulse train fault model and a random wave train fault model. In each fault model, we have specified the kind of signals that can be generated by failed modules as well as the interaction between such signals and fault-free hardware elements in fault handlers. These fault models provide a vigorous basis for studying fault-tolerant hardware implementations.

Redundancy Management

We have developed an approach to coordinate byte slices in each redundant module so that failures among these slices can be detected and/or masked by decoding the outputs they generate. This approach is based on maintaining timing synchronization and consistency among byte slices in the same module. This approach to redundancy management is supported by two robust algorithms under which timing synchronization and consistency can be maintained even after certain hardware failures have occurred. The effectiveness of this approach in dealing with hardware failures characterized by the stuck-at fault model, the random pulse train fault model and the random wave train fault model is also investigated. We have also briefly discussed an alternative approach to redundancy management for fault tolerance, based on buffering schemes, and its relative merits and disadvantages.

We have presented control module designs which can detect and/or mask hardware failures under the stuck-at fault model without ever generating runt pulses. Under the random pulse train fault model, these control modules can generate runt pulses due to metastable state phenomena, but the probability of such occurrences can be reduced to acceptable levels. We have also outlined an approach to deal with random wave train faults, but have not analyzed this approach in sufficient detail to evaluate its effectiveness.

Our timing synchronization technique is closely related to the synchronization voting technique studied by Davies and Wakerly [17], and has applications in fault-tolerant clock design. We have clarified the problems in this area and their solution techniques through investigating the hardware implementation of our timing synchronization technique under the different fault models.

7.1.2 Fault-Tolerant Data Flow Processor Design

A data flow processor can be rendered fault-tolerant by implementing a fault masking capability in each of its hardware modules. As an alternative approach we have developed a fault tolerance strategy for masking hardware failures in the data flow processor based on dynamic redundancy. This alternative approach offers the potential of considerable hardware savings, especially in the packet networks.

The data flow processor is a parallel processing system in which several machine instructions may be executed in parallel, and intermodule communication is supported by packet networks. The STAR computer [8] is probably the first fault-tolerant computer organized by a dynamic redundancy scheme. In this computer fault detection is supported by using arithmetic codes in arithmetic units and system buses, and duplication in units that perform logic operations. Spare hardware modules and spare bus lines are incorporated and switched in to replace failed modules after hardware failures are detected and diagnosed. The STAR computer executes machine instructions serially, and state information is saved at program checkpoints for subsequent rollbacks to recover from detected failures after repair. Our dynamic redundancy scheme can be considered a variant of the strategy implemented on the STAR computer, refined in accordance with the architectural characteristics of the data flow computer:

- Failures in processing elements, specialized functional units, routers and allocators are detected by designing these modules, and encoding packet nibbles, to support parity checks.
- Spare routers and allocators are embedded in packet networks to support rapid repair. Homogeneous sets of processing elements and specialized functional units are provided so that the system can degrade gracefully when these modules fail.
- Transmitted packets are saved until they are acknowledged, and retransmitted upon request.

The dynamic redundancy scheme is developed by first focusing on fault tolerance requirements for the packet networks. A system strategy for fault tolerance is then developed based on the selected requirements. These requirements are also used to derive specifications for hardware modules. The desired fault tolerance capabilities are implemented in these hardware modules using hardware redundancy techniques developed in the first part of this thesis. This same methodology can be used to design other dynamic redundancy schemes for fault-tolerant data flow processors, using different fault tolerance requirements for the networks.

Our design is but one of many possible alternatives for constructing fault-tolerant data flow processors. It nonetheless demonstrates that fault tolerance can be incorporated into a high performance computing system with no impact on its programmability.

7.1.3 Evaluation

The data flow processor is designed to achieve high performance through parallel processing. This parallel processing capability is not compromised in the fault-tolerant processor design we have proposed. Individual hardware operations may take longer to perform due to input decoding and synchronization. But pipelined and concurrent operation of hardware modules are still supported. In other words, fault tolerance mechanisms incorporated according to our strategy lengthens the execution time of microscopic operations, but has no effect on concurrency exploitation at higher levels.

For our intended applications, we also favor a self-timed implementation approach guided by packet communication principles. In constructing a totally self-timed system, the timing characteristics of individual modules need not be known. In our approach to redundancy management, we have relied upon performance compatibility among byte slices in the same redundant module to maintain proper synchronization, and we have also made assumptions on gate

delays and path delays in control module designs. For each redundant module, however, the only timing characteristics which must be made available to its environment for its successful deployment are phase differences among packets and acknowledgments in the same output batch generated by fault-free slices in that module. In synchronous systems, upper bounds on execution times for basic hardware operations in each module must also be calculated. Our procedure for integrating redundant packet communication modules into fault-tolerant packet communication systems is thus somewhat more complex than that for self-timed systems, and yet considerably simpler than that for synchronous systems. We have lost some, but not all, of the desirable modularity properties of self-timed systems in our fault tolerance methodology.

We have shown how to construct redundant packet communication systems so that hardware failures limited to some maximum number of byte slices in each redundant module can be tolerated. We have, however, not addressed the issue of evaluating the reliability improvements attainable through such enhancements. We have developed a design methodology for incorporating redundant hardware into packet communication systems for fault tolerance, a set of mechanisms to support this methodology, and illustrated its application in a specific system design. Alternative approaches to redundant management, alternative mechanisms to support these approaches, and alternative system designs can be developed to guide the construction of fault-tolerant parallel processing systems based on self-timed principles. When detailed logic designs and hardware failure rates are available, alternative schemes can be evaluated more carefully to compare their reliability properties and cost-effectiveness. In this thesis, we have identified the technical issues and laid the foundation on which such alternatives can be developed and examined.

7.2 Suggestions for Further Research

We have presented a design methodology for constructing fault-tolerant packet communication systems and a system design for a fault-tolerant data flow processor. Let us first consider some alternative solutions to redundancy management and packet network design in this framework, and then discuss how our concepts and techniques can be applied to achieve fault-tolerance in more general forms of data flow processors and in constructing fault-tolerant computers for other applications.

Redundancy Management

Our approach to redundancy management is based on maintaining timing synchronization and consistency in redundant systems. An alternative approach to maintaining timing synchronization is described briefly in Section 5.5. This approach is based on the idea of buffering packets and acknowledgments in control modules until they can be forwarded. It is conceptually simpler but implementation details must be worked out before its hardware requirements and fault tolerance capabilities under different fault models can be evaluated. Methods for determining buffer sizes in control modules and strategies for dealing with buffer overflow must also be established.

For our timing synchronization algorithm, $3f+1$ synchronizers are needed to tolerate failures in up to f synchronizers among them. The question of whether there exists robust timing synchronization algorithms which can be implemented with fewer synchronizers, or whether $3f+1$ is a lower bound for solving this problem remains unsettled. We note that under restricted fault assumptions, timing synchronization can be maintained using synchronization voting [17], which does not require any synchronization among byte slices in a redundant module.

We have also suggested an approach to implement the timing synchronization algorithm in control modules under the random wave train fault model. In this approach we propose to use lowpass filters to eliminate random wave trains of bounded wavelength. This approach should be investigated using analog circuit design and analysis techniques to establish methods for dealing with random wave train faults.

Packet Network Design

Hardware failures in packet networks can be masked by implementing a fault masking capability, based on either error correcting codes or replication, in every router and allocator. We have adopted an alternative approach under which some hardware failures are masked and others are only detected. Other alternatives include relying entirely on packet encoding techniques, such as adding a checksum nibble to each packet, to detect failures, and using time-out mechanisms to control retransmission, as in the ARPAnet. It is worthwhile to conduct a more systematic study of fault-tolerant network design, classifying different techniques according to the implementation technologies and failure modes for which they are effective.

We have also described two strategies for incorporating redundant hardware into a rectangular packet network to support rapid repair. Interconnection networks in large parallel processing systems often have regular structures and are constructed out of a large number of identical units. There are thus many opportunities for exploiting structural regularity and uniformity in these networks in incorporating redundant modules and paths into them.

Redundant Data Flow Computer Systems

We have presented the design of a fault-tolerant data flow processor intended for numerical computation applications. This processor supports only a subset of the data flow language presented

in [22]. More general forms of data flow computer systems [1], [65] must be developed to support environments suitable for data base applications and general purpose computing. These systems should also be fault-tolerant to improve availability and maintainability.

Redundant Microcomputer Systems

Many real-time control applications can benefit from the availability of low-cost fault-tolerant computer systems. A redundant system can be constructed using several independently clocked microcomputers, and maintaining timing synchronization and consistency among them. Redundancy management can be implemented in hardware, by incorporating the suitable hardware mechanisms in the microcomputer chip, or in software, by incorporating the corresponding operations into application programs. It is a challenging research problem to design a fault-tolerant microcomputer system, and the corresponding programming methodology, for real-time control applications based on these ideas.

Finally, we note that many engineering decisions must be made in applying the methodology explained in this thesis to the construction of practical fault-tolerant systems. Such decisions concern system modularization for redundancy incorporation, the choice of a suitable fault model for the implementation technology, and the choice of redundancy techniques subjected to reliability, performance and cost requirements. Methodologies can also be developed for making such decisions.

References

[1] W. B. Ackerman, "*A structure memory for data flow computers*", Laboratory for Computer Science, Massachusetts Institute of Technology, TR-186, August, 1977.

[2] W. B. Ackerman, "Data Flow Languages", *Proceedings of the 1979 National Computer Conference*, pp. 1087-1095, June 1979.

[3] W. B. Ackerman and J. B. Dennis "*VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*", Laboratory for Computer Science, Massachusetts Institute of Technology, TR-218, June 1979.

[4] D. B. Armstrong, A. D. Friedman and P. R. Menon. "Design of asynchronous circuits assuming unbounded gate delay", *IEEE-TC* C-18, 12. pp 1110 - 1120, Dec. 1969.

[5] D. B. Armstrong, A. D. Friedman and P. R. Menon. "Design of asynchronous circuits assuming unbounded gate delay", Bell Telephone Labs. internal memorandum (unpublished).

[6] D. A. Anderson. *Design of self-checking digital networks using coding techniques*, Coordinated Science Laboratory Rep. R-527, University of Illinois, Urbana, Oct. 1971.

[7] A. Avizienis, "Fault-tolerance: the survival attribute of digital systems", *Proc IEEE*, vol 66, 10, pp. 1109 - 1125, Oct 1978.

[8] A. Avizienis, et al., "The STAR (Self-Testing-And-Repairing) computer: An investigation of the theory and practice of fault-tolerant computer design", *IEEE-TC*, vol. C-20, 11, pp. 1312-1321, Nov. 1971.

[9] A. Avizienis, "Fault-tolerance and longevity: goals for high-speed computers of the future", *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*. Academic Press, pp. 173-178, 1977.

[10] A. Avizienis, M. Ercegovac, T. Lang, P. Sylvain & A. Thomasian "An investigation of fault-tolerant architectures for large-scale numerical computing", *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*. Academic Press, pp. 159-171, 1977.

[11] B. R. Borgerson, "Spontaneous reconfiguration in a fail-softly computer utility", *Proc. Datafair 73*, British Computer Society, London, pp 326-333, 1973.

[12] G. A. Boughton, *Routing Networks in Packet Communication Architectures*, Dept. of Electrical Engineering and Computer Science, M.I.T., S.M. Thesis, June 1978.

[13] W. C. Carter and P. R. Schneider. "Design of dynamically checked computers", IFIP Congr.68, vol. 2, pp 878 - 883, Edinburgh, Scotland 1968.

[14] T. J. Chaney, S. M. Ornstein, and W. M. Littlefield, "Beware the Synchronizer", *Digest of Papers - CompCon '72*, IEEE, pp. 317-319, Sept., 1972.

[15] T. J. Chaney & C. E. Molnar. "Anomalous behavior of synchronizer and arbiter circuits", *IEEE-TC* C-22, 4, pp 421-422, April 1973.

[16] W. M. Daly, A. L. Hopkins and J. F. McKenna. "A fault-tolerant digital clocking system", *Dig. 3rd Int. Symp. Fault-Tolerant Comp.* Palo Alto, CA, pp 17 - 22, June 1973.

[17] D. Davies and J. F. Wakerly. "Synchronization and matching in redundant systems", *IEEE-TC* vol. C-27, 6, pp 531 - 539, June 1978.

[18] D. Davies. "Reliable synchronization of redundant systems", *Proc. 5th Annual Symp. Comp. Arch* Palo Alto, CA. April 3 - 5, 1978.

[19] D. Davies, "Reliable synchronization of redundant systems", Internal Memo, Digital Systems Laboratory, Stanford University, Stanford, California, Oct., 1977.

[20] J. B. Dennis, "The varieties of data flow computers", *Proceedings of the 1st International Conference on Distributed Systems*, IEEE, October 1979.

[21] J. B. Dennis "Packet Communication Architecture", *Proc. 1975 Sagamore Computer Conference on Parallel Processing*, Syracuse University, August 1975.

[22] J. B. Dennis, "First version of a data flow procedural language", *Lecture Notes in Computer Science*, vol. 19, New York: Springer-Verlag, pp. 362-376, 1974.

[23] J. B. Dennis, "Computation Structures", *COSINE Committee Lectures*, Princeton University, Department of Electrical Engineering, Princeton, New Jersey, July 1968.

[24] J. B. Dennis, G.A. Boughton & C.K.C. Leung "Building blocks for data flow prototypes" to be presented at the 7th Annual Symposium on Computer Architecture, France, May 1980.

[25] J. B. Dennis, & D.P. Misunas "A preliminary architecture for a basic data-flow processor." *Proceedings of the 2nd Annual Symposium on Computer Architecture*, IEEE, New York, pp. 126-132, 1975.

[26] J. B. Dennis, C.K.C. Leung & D.P. Misunas "A highly parallel processor using a data flow machine language", M.I.T. Laboratory for Computer Science, Computation Structures Group, Memo 134-1, Cambridge, Mass, also to appear in IEEE Transaction on Computers.

[27] D. J. Ellis "*Formal Specifications for Packet Communication Systems*", Laboratory for Computer Science, Massachusetts Institute of Technology, TR-189, November 1977.

[28] J. Galiay, Y. Crouzet and M. Vergniault "Physical versus logical fault models in MOS LSI circuits, impact on their testability", *Proc. of the 1979 Symp. on Fault-Tolerant Computing*, IEEE, pp 195-202, June, 1979.

[29] R. W. Hamming, "Error detecting and error correcting code", *Bell System Technical Journal*, vol. 29, no. 2, pp 147-160, April 1950.

[30] A. L. Hopkins, Jr., T. B. Smith, III, and J. H. Lala "FTMP - A highly reliable fault-tolerant multiprocessor for aircraft", *Proc. IEEE*, vol 66, 10, pp 1221 - 1239, October 1978.

[31] R. M. Keller. "Towards a theory of universal speed-independent modules", *IEEE-TC* C-23, 1, pp 21 - 33, Jan 1974.

[32] P. R. Kosinski "*Denotational Semantics of Determinate and Non-Determinate Data Flow Programs*", Laboratory for Computer Science, Massachusetts Institute of Technology, TR-220, May 1979.

[33] C. K. C. Leung *On a Design Methodology for Packet Communication Systems based on a Hardware Design Language*, CSG Group Memo under preparation, Lab. for Computer Science, MIT, August 1979.

[34] C. K. C. Leung "ADL, an architecture description language for packet communication systems", CSG Group Memo, Lab. for Computer Science, MIT August, 1979, also presented at *5th Symp on CHDL*, Palo Alto, Ca, Oct. 1979.

[35] K. N. Levitt, M.W. Green, & J. Goldberg, "A study of data commutation problems in a self-repairable multiprocessor", *AFIPS Conference Proceedings, vol. 32*, (1968 SJCC), Thompson Book Company, Washington, D.C, 1968, pp. 515-527.

[36] L. R. Marino "The effect of asynchronous inputs on sequential network reliability", *IEEE-TC*, vol C-26, 11, pp 1082 - 1090, Nov. 1977.

[37] D. P. Misunas, "Deadlock avoidance in a data flow architecture", *Proceedings of the Milwaukee Symposium on Automatic Computation and Control*, IEEE, April 1975.

[38] D. E. Muller "Asynchronous logics and application to information processing", *Switching theory in Spacer Technology* Stanford University Press, Stanford, CA 1963.

[39] A. Newell and H. A. Simon, "Computer science as empirical inquiry: symbols and search", *Communications of the ACM*, vol. 19, no. 3, pp. 113-126, Mar. 1976.

[40] Y. W. Ng and A. Avizienis, "A reliability model for gracefully degradable and repairable fault-tolerant systems", *Proc. 1977 Int. Symp. Fault-Tolerant Computing*, IEEE, pp. 22-28, June 1977.

[41] B. Parhami and A. Avizienis, "A study of fault tolerance techniques for associative processors", *Proc. NCC*, pp. 643-652, 1974.

[42] S. S. Patil *Bounded and unbounded delay synchronizers and arbiters*, Computation Structures Group Memo 103, Laboratory for Computer Science, M.I.T., Cambridge, Mass., June 1974.

[43] M. Pechoucek, "Anomalous response times of input synchronizers", IEEE-TC, vol. C-25, 2, pp. 133-139, Feb., 1976.

[44] M. Pease, R. Shostak and L. Lamport "Reaching agreement in the presence of faults", *Jacm*, vol. 27, no. 2, pp. 228-234, Apr., 1980.

[45] W. W. Peterson and E. J. Weldon, *Error-Correcting Codes*, MIT Press, Cambridge, Mass., 1971.

[46] T. R. N. Rao, *Error Coding for Arithmetic Processors*, Academic Press, New York, New York, 1974.

[47] C. L. Seitz "System Timing", Chapter 7 in *Introduction to VLSI Systems*, by C. A. Mead and L. A. Conway, Addison-Wesley Press, October 1979.

[48] C. L. Seitz "Self-Timed VLSI Systems", *Proceedings of the Caltech Conference on VLSI*, Pasadena, California, January 1979.

[49] Sessions on Data Flow Computer Architectures. Proc. of AFIPS Conference, 1979.

[50] Sessions on Data Flow Computer Architectures. Proc. of Compcon 80, IEEE, Feb. 1980.

[51] Session on the Fault-Tolerant Spaceborne Computer. *Proc. of the 1976 Int. Symp. on Fault-Tolerant Computing*, pp. 129-147, Pittsburg, PA, June, 1976.

[52] Special issue on fault-tolerant digital systems, Proc. IEEE, vol. 66, 10, October 1978.

[53] Special issue on fault-tolerant computing, Computer, vol. 4, 1, Jan/Feb 1971.

[54] Special issue on fault-tolerant computing, Computer, vol. 13, 3, March 1980.

[55] C. Sheridan, "Space Shuttle Software", *Datamation*, July 1978.

[56] T. B. Smith, III, "A damage- and fault-tolerant input/output network", *IEEE-TC*, vol. C-24, no. 5, pp. 505-512, May 1975.

[57] J. J. Stiffler, N. G. Parke IV, and P. C. Barr, "The SERF fault-tolerant computer", Parts I and II, *Proc. of the 1973 Int. Symp. Fault-Tolerant Computing*, Palo Alto, CA, June, 1973.

[58] W. N. Toy, "Fault-tolerant design of local ESS Processors", *Proc. IEEE*, pp. 1126-1145, Oct. 1978.

[59] A. R. Tripathi and G. J. Lipovski, "Packet Switching in Banyan Networks", *Proceedings of the 6th Annual Symposium on Computer Architecture*, IEEE, pp. 160-167, April 1979.

[60] J. G. Tyron, "Quadded logic", *Redundancy Techniques for Computing Systems*, Spartan Books, pp. 205-228, 1962.

[61] E. Vishniac, "*A processor module for data flow computer development*", Laboratory for Computer Science, M.I.T., CSG Memo 176, May 1979.

[62] R. L. Wadsack, *Fault modeling and logic simulation of CMOS and MOS integrated circuits*, Bell System Technical Journal, vol. 57, 5, pp 1449-1473, May-June 1978.

[63] J. F. Wakerly, "Transient failures in triple modular redundancy systems with sequential modules", *IEEE-TC*, vol. 24, 5, pp. 570-573, May 1975.

[64] J. F. Wakerly *Error-detecting codes, self-checking circuits and applications*, Elsevier-North Holland, New York 1978.

[65] K. S. Weng *An abstract implementation of a generalized data flow language*", Laboratory for Computer Science, Massachusetts Institute of Technology, Technical Report TR-228, 1980.

[66] 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 - The design and analysis of a fault-tolerant computer for aircraft control", *Proc IEEE*, vol 66, 10, pp 1240 - 1255, Oct 1978

[67] W. A. Wulf and C. G. Bell "C.mmp - a multi-mini-processor", *Proc. AFIPS 1972*, FJCC. vol. 41, AFIPS Press, Montvale, NJ, pp. 765-777.

Biographical Note

Clement Kin Cho Leung was born in Hong Kong on December 26, 1949. He received his high school education at Wah Yan College, Hong Kong. He was awarded a Graham Scholarship for his achievements in the Hong Kong School Certificate Examination in 1968.

From 1968 to 1980, he was enrolled at the Massachusetts Institute of Technology. He received the degrees of Bachelor of Science, Master of Science and Electrical Engineer from the Department of Electrical Engineering and Computer Science in June, 1975. He also held an instructor appointment in the Department of Electrical Engineering and Computer Science from 1975 to 1978.

As a member of the Computation Structures Group at the Laboratory for Computer Science at MIT, Mr. Leung has conducted research in parallel processing, data flow computer architecture, architecture description languages, and fault-tolerant computer design. He is also a member of the IEEE and the ACM, and has been elected to Tau Beta Pi, Eta Kappa Nu, and Sigma Xi.

Mr. Leung is currently a research associate at the Laboratory for Computer Science at MIT. He is married to Enid Yee Wan Yim.