# An Introduction to the Design and Analysis of Fault-Tolerant Systems

Barry W. Johnson

Department of Electrical Engineering
Center for Semicustom Integrated Systems
University of Virginia
Charlottesville, Virginia 22903-2442

Portions of this material are adapted from the textbook *Design and Analysis of Fault-Tolerant Digital Systems*, by Barry W. Johnson, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

# Table of Contents

## 1.1. Introduction

This chapter is devoted to the study of introductory concepts and techniques for designing and analyzing fault-tolerant systems. A *fault-tolerant system* is one that can continue the correct performance of its specified tasks in the presence of hardware and/or software faults. *Fault tolerance* is the attribute that enables a system to achieve fault-tolerant operation. Finally, the term *fault-tolerant computing* is used to describe the process of performing calculations, such as those performed by a computer, in a fault-tolerant manner. This chapter is divided into four primary sections. The remainder of this first section is an introduction to the basic terminology used in the fault tolerance field. The second section is an overview of redundancy techniques employed in the design of fault-tolerant systems. Hardware, software, time, and information redundancy methods are considered. The third section describes evaluation metrics such as reliability, availability, and safety, as well as the modeling techniques used to determine numerical values for these metrics. Finally, the last section provides an overview of the design process and the incorporation of fault avoidance ideas. The material in this chapter is intended to provide an introduction to the various topics important to the fault tolerance field. After completing this chapter the reader should be able to pursue each of the individual topics in more detail.

### 1.1.1. Fundamental Terminology

Three fundamental terms in fault-tolerant design are fault, error, and failure. There is a cause-effect relationship between faults, errors, and failures. Specifically, faults are the cause of errors, and errors are the cause of failures. Often the term failure is used interchangeably with the term malfunction, however, the term failure is rapidly becoming the more commonly accepted one.

A *fault* is a physical defect, imperfection, or flaw that occurs within some hardware or software component. Essentially, the definition of a fault, as used in the fault tolerance community, agrees with the definition found in the dictionary. A fault is a blemish, weakness, or shortcoming of a particular hardware or software component. An *error* is the manifestation of a fault. Specifically, an error is a deviation from accuracy or correctness. Finally, if the error results in the system performing one of its functions incorrectly then a system *failure* has occurred. Essentially, a failure is the nonperformance of some action that is due or expected. A failure is also the performance of some function in a subnormal quantity or quality.

The concepts of faults, errors, and failures can be best presented by the use of a three-universe model that is an adaptation of the four-universe model originally developed in [AVIZIENIS 82]. The first universe is the *physical universe* in which faults occur. The physical universe contains the semiconductor devices, mechanical elements, displays, printers, power supplies, and other physical entities that make up a system. A fault is a physical defect or alteration of some component within the physical universe. The second universe is the *informational universe*. The informational universe is where the error occurs. Errors affect units of information such as data words within a computer or digital voice or image information. An error has occurred when some unit of information becomes incorrect. The final universe is the *external* or *user's universe*. The external universe is where the user of a system ultimately sees the effect of faults and errors. The external universe is where failures occur. The failure is any deviation that occurs from the desired or expected behavior of a system. In summary, faults are physical events that occur in the physical universe. Faults can result in errors in the informational universe, and errors can ultimately lead to failures that are witnessed in the external universe of the system.

The cause-effect relationship implied in the three-universe model leads to the definition of two important parameters; fault latency and error latency. *Fault latency* is the length of time between the occurrence of a fault and the appearance of an error due to that fault. *Error latency* is the length of time between the occurrence of an error and the appearance of the resulting failure. Based on the three-universe model, the total time between the occurrence of a physical fault and the appearance of a failure will be the sum of the fault latency and the error latency.

Faults can be the result of a variety of things that occur within electronic components, external to the components, or during the component or system design process. Problems at any of several points within the design process can result in faults within the system. At the highest level is the possibility of *specification mistakes*, which include incorrect algorithms, architectures, or hardware and software design specifications. The next cause of faults is *implementation mistakes*. Implementation, as defined here, is the process of transforming hardware and software specifications into the physical hardware and the actual software. The implementation can introduce faults because of poor design, poor component selection, poor construction, or software coding mistakes. The next cause of faults is *component defects*. Manufacturing imperfections, random device

defects, and component wear-out are typical examples of component defects. Electronic components simply become defective sometimes. The defect can be the result of bonds breaking within the circuit or corrosion of the metal. Component defects are the most commonly considered cause of faults. The final cause of faults is the *external disturbance*; for example, radiation, electromagnetic interference, battle damage, operator mistakes, and environmental extremes.

To adequately describe faults, characteristics other than the cause are required. In addition to the cause, four major attributes are critical to the description of faults; nature, duration, extent, and value [NELSON 82]. The *nature* of a fault specifies the type of fault; for example, whether it is a hardware fault, a software fault, a fault in the analog circuitry, or a fault in the digital circuitry. Another key attribute of a fault is its *duration.* The duration specifies the length of time that a fault is active. First, there is the *permanent fault,* that remains in existence indefinitely if no corrective action is taken. Second, there is the *transient fault,* that can appear and disappear within a very short period of time. Third, there is the *intermittent fault* that appears, disappears, and then reappears repeatedly. The next attribute of faults is the *extent.* The extent of a fault specifies whether the fault is localized to a given hardware or software module or globally affects the hardware, the software, or both. The final attribute of faults is the *value.* The value of a fault can be either determinate or indeterminate. A *determinate fault* is one whose status remains unchanged throughout time unless externally acted upon. An *indeterminate fault* is one whose status at some time, T, may be different from its status at some increment of time greater than or less than T.

There are three primary techniques for attempting to improve or maintain a system's normal performance in an environment where faults are of concern; fault avoidance, fault masking, and fault tolerance. *Fault avoidance* is a technique that is used in an attempt to prevent the occurrence of faults. Fault avoidance can include such things as design reviews, component screening, testing, and other quality control methods. *Fault masking* is any process that prevents faults in a system from introducing errors into the informational structure of that system. Finally, *fault tolerance* is the ability of a system to continue to perform its tasks after the occurrence of faults. The ultimate goal of fault tolerance is to prevent system failures from occurring. Since failures are directly caused by errors, the terms *fault tolerance* and *error tolerance* are often used interchangeably.

Fault tolerance can be achieved by many techniques. Fault masking is one approach to toler-

ating faults. Another approach is to detect and locate the fault and reconfigure the system to remove the faulty component. *Reconfiguration* is the process of eliminating a faulty entity from a system and restoring the system to some operational condition or state. If the reconfiguration technique is used then the designer must be concerned with fault detection, fault location, fault containment, and fault recovery. *Fault detection* is the process of recognizing that a fault has occurred. Fault detection is often required before any recovery procedure can be implemented. *Fault location* is the process of determining where a fault has occurred so that an appropriate recovery can be implemented. *Fault containment* is the process of isolating a fault and preventing the effects of that fault from propagating throughout a system. Fault containment is required in all fault-tolerant designs. Finally, *fault recovery* is the process of remaining operational or regaining operational status via reconfiguration even in the presence of faults.

Equivalent definitions can be provided in the informational universe. Specifically, *error detection* is the process of recognizing that an error has occurred. *Error location* is the process of determining which specific module produced the error. *Error containment* is the process of preventing the error from propagating throughout a system. Finally, *error recovery* is the process of regaining operational status or restoring the system's integrity after the occurrence of an error.

### 1.1.2. Objectives of Fault Tolerance

Fault tolerance is an attribute that is designed into a system to achieve some design goal. Just as a design must meet many functional and performance goals, it must also satisfy numerous other requirements as well. The most prominent of the additional requirements are dependability, reliability, availability, safety, performability, maintainability, and testability; fault tolerance is one system attribute capable of fulfilling such requirements.

**Dependability**. The term *dependability* is used to encapsulate the concepts of reliability, availability, safety, maintainability, performability, and testability. Dependability is simply the quality of service provided by a particular system [LAPRIE 85]. Reliability, availability, safety, maintainability, performability, and testability, are examples of measures used to quantify the dependability of a system.

**Reliability**. The *reliability* of a system is a function of time, *R(t)*, defined as the conditional

probability that the system performs correctly throughout the interval of time, $[t_0,t]$, given that the system was performing correctly at time $t_0$. In other words, the reliability is the probability that the system operates correctly throughout a complete interval of time. The reliability is a conditional probability in that it depends on the system being operational at the beginning of the chosen time interval. The *unreliability* of a system is a function of time, *F(t)*, defined as the conditional probability that a system begins to perform incorrectly during the interval of time, $[t_0,t]$, given that the system was performing correctly at time $t_0$. The unreliability is often referred to as the probability of failure.

Reliability is most often used to characterize systems in which even momentary periods of incorrect performance are unacceptable, or it is impossible to repair the system. If repair is impossible, such as in many space applications, the time intervals being considered can be extremely long, perhaps as many as ten years. In other applications, such as aircraft flight control, the time intervals of concern may be no more than several hours, but the probability of working correctly throughout that interval may be 0.9999999 or higher. It is a common convention when reporting reliability numbers to use $0.9_i$ to represent the fraction that has *i* nines to the right of the decimal point. For example, 0.9999999 is written as $0.9_7$.

**Availability**. *Availability* is a function of time, *A(t)*, defined as the probability that a system is operating correctly and is available to perform its functions at the instant of time, *t*. Availability differs from reliability in that reliability involves an interval of time, while availability is taken at an instant of time. A system can be highly available yet experience frequent periods of inoperability as long as the length of each period is extremely short. In other words, the availability of a system depends not only on how frequently it becomes inoperable but also how quickly it can be repaired. Examples of high-availability applications include time-shared computing systems and certain transactions processing applications, such as airline reservation systems.

**Safety**. *Safety* is the probability, *S(t)*, that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the operation of other systems or compromise the safety of any people associated with the system. Safety is a measure of the fail-safe capability of a system; if the system does not operate correctly, it is desired to have the system fail in a safe manner. Safety and reliability differ because reliability is the probability that a system

will perform its functions correctly, while safety is the probability that a system will either perform its functions correctly or will discontinue the functions in a manner that causes no harm.

**Performability**. In many cases, it is possible to design systems that can continue to perform correctly after the occurrence of hardware and software faults, but the level of performance is somehow diminished. The *performability* of a system is a function of time, *P(L,t)*, defined as the probability that the system performance will be at, or above, some level, *L*, at the instant of time, *t* [FORTES 84]. Performability differs from reliability in that reliability is a measure of the likelihood that all of the functions are performed correctly, while performability is a measure of the likelihood that some subset of the functions is performed correctly.

Graceful degradation is an important feature that is closely related to performability. *Graceful degradation* is simply the ability of a system to automatically decrease its level of performance to compensate for hardware and software faults. Fault tolerance can certainly support graceful degradation and performability by providing the ability to eliminate the effects of hardware and software faults from a system, therefore allowing performance at some reduced level.

**Maintainability.** *Maintainability* is a measure of the ease with which a system can be repaired, once it has failed. In more quantitative terms, maintainability is the probability, *M(t)*, that a failed system will be restored to an operational state within a period of time *t*. The restoration process includes locating the problem, physically repairing the system, and bringing the system back to its operational condition. Many of the techniques that are so vital to the achievement of fault tolerance can be used to detect and locate problems in a system for the purpose of maintenance. Once the problem is located, maintenance can then be performed to implement the necessary repairs. Automatic diagnostics can significantly improve the maintainability of a system because a majority of the time used to repair a system is often devoted to determining the source of the problem.

**Testability**. *Testability* is simply the ability to test for certain attributes within a system. Measures of testability allow one to assess the ease with which certain tests can be performed. Certain tests can be automated and provided as an integral part of the system to improve the testability. Many of the techniques that are so vital to the achievement of fault tolerance can be used to detect

and locate problems in a system for the purpose of improving testability. Testability is clearly related to maintainability because of the importance of minimizing the time required to identify and locate specific problems.

### 1.1.3. Applications of Fault-Tolerant Computing

Applications of fault-tolerant computing can be categorized into four primary areas; long-life applications, critical computations, maintenance postponement, and high availability. Each application presents differing design requirements and challenges.

**Long-life Applications**. The most common examples of long-life applications are the unmanned space flight and satellites. Typical requirements of a long-life application are to have a 0.95, or greater, probability of being operational at the end of a ten-year period. Unlike other applications, however, long-life systems can often allow extended outages as long as the system can eventually be made operational once again. In addition, long-life applications can frequently allow the system to be reconfigured manually by the operators. The Fault-Tolerant Space-borne Computer (FTSC) [STIFFLER 76], the Self-Testing And Repairing (STAR) computer [AVIZIENIS 71b], and the Fault-Tolerant Building Block Computer (FTBBC) [RENNELS 80] are classical examples of systems designed for long-life applications.

**Critical-computation Applications**. Perhaps the most widely-publicized applications of fault-tolerant computing are those where the computations are critical to human safety, environmental cleanliness, or equipment protection. Examples include aircraft flight control systems, military systems, and certain types of industrial controllers. In critical-computation applications, the incorrect performance of the system will almost certainly yield devastating results. A typical requirement for a critical-computation application is to have a reliability of $0.9_7$ at the end of a three-hour time period. Requirements can vary, however, depending on the particular function that the system is performing. One of the most publicly visible critical-computation application of fault-tolerant computing has been the space shuttle [SKLAROFF 76]. Industrial control systems also perform critical computations. For example, chemical reactions may have to be precisely controlled to prevent explosions or other unwanted effects.

**Maintenance Postponement Applications**. Maintenance postponement applications appear

most frequently when maintenance operations are extremely costly, inconvenient, or difficult to perform. Remote processing stations and certain space applications are good examples. In space, maintenance can be impossible to perform, while at remote sites the cost of unexpected maintenance can be prohibitive. The main goal is to use fault tolerance to allow maintenance to be postponed to more convenient and cost-effective times. Maintenance personnel can visit a site monthly and perform any necessary repairs. Between maintenance visits, the system uses fault tolerance to continue to perform its tasks. A telephone switching system [TOY 78] is an example where maintenance postponement may be required. Many telephone switching systems are located in remote areas where it is necessary to provide telephone service, but it is costly to perform the maintenance and service operations. The primary objective is to design the system such that unscheduled maintenance can be avoided. Therefore, the telephone company can visit the facility periodically and repair the system or perform routine maintenance. Between maintenance visits, the system handles failures and service disruptions autonomously.

**High Availability Applications**. Availability is a key parameter in many applications. Banking and other time-shared systems are good examples of high availability applications. Users of these systems want to have a high probability of receiving service when it is requested. The Tandem Nonstop transaction processing system [KATZMAN 77] is a good example of one designed for high availability.

## 1.2. Redundancy Techniques

The concept of *redundancy* implies the addition of information, resources, or time beyond what is needed for normal system operation. The redundancy can take one of several forms, including hardware redundancy, software redundancy, information redundancy, and time redundancy. The use of redundancy can provide additional capabilities within a system. In fact, if fault tolerance or fault detection is required then some form of redundancy is also required. But, it must be understood that redundancy can have a very important impact on a system in the areas of performance, size, weight, power consumption, reliability, and others.

### 1.2.1. Hardware Redundancy

The physical replication of hardware is perhaps the most common form of redundancy used in sys-

tems. As semiconductor components have become smaller and less expensive, the concept of hardware redundancy has become more common and more practical. The costs of replicating hardware within a system are decreasing simply because the costs of hardware are decreasing.

There are three basic forms of hardware redundancy. First, *passive* techniques use the concept of fault masking to hide the occurrence of faults and prevent the faults from resulting in errors. Passive approaches are designed to achieve fault tolerance without requiring any action on the part of the system or an operator. Passive techniques, in their most basic form, do not provide for the detection of faults but simply mask the faults.

The second form of hardware redundancy is the *active* approach, which is sometimes called the *dynamic* method. Active methods achieve fault tolerance by detecting the existence of faults and performing some action to remove the faulty hardware from the system. In other words, active techniques require that the system perform reconfiguration to tolerate faults. Active hardware redundancy uses fault detection, fault location, and fault recovery in an attempt to achieve fault tolerance.

The final form of hardware redundancy is the *hybrid* approach. Hybrid techniques combine the attractive features of both the passive and active approaches. Fault masking is used in hybrid systems to prevent erroneous results from being generated. Fault detection, fault location, and fault recovery are also used in the hybrid approaches to improve fault tolerance by removing faulty hardware and replacing it with spares. Providing spares is one form of providing redundancy in a system. Hybrid methods are most often used in the critical-computation applications where fault masking is required to prevent momentary errors, and high reliability must be achieved. Hybrid hardware redundancy is usually a very expensive form of redundancy to implement.

**Passive Hardware Redundancy.** Passive hardware redundancy relies upon voting mechanisms to mask the occurrence of faults. Most passive approaches are developed around the concept of majority voting. As previously mentioned, the passive approaches achieve fault tolerance without the need for fault detection or system reconfiguration; the passive designs inherently tolerate the faults.

The most common form of passive hardware redundancy is called *triple modular redundancy*

(TMR). The basic concept of TMR is to triplicate the hardware and perform a majority vote to determine the output of the system. If one of the modules becomes faulty, the two remaining fault-free modules mask the results of the faulty module when the majority vote is performed. The basic concept of TMR is illustrated in Figure 1.1. In typical applications, the replicated modules are processors, memories, or any hardware entity. A simple example of TMR is shown in Figure 1.2 where data from three independent processors is voted upon before being written to memory. The majority vote provides a mechanism for ensuring that each memory contains the correct data, even if a single faulty processor exists. A similar voting process is provided at the output of the memories, so that a single memory failure will not corrupt the data provided to any one processor. Note that in Figure 1.2 there are three separate voters so that the failure of a single voter cannot corrupt more than one memory or more than one processor.

The primary difficulty with TMR is obviously the voter; if the voter fails, the complete system fails. In other words, the reliability of the simplest form of TMR, as shown in Figure 1.1, can be no better than the reliability of the voter. Any single component within a system whose failure will lead to a failure of the system is called a *single-point-of-failure*. Several techniques can be used to overcome the effects of voter failure. One approach is to triplicate the voters and provide three independent outputs, as illustrated in Figure 1.2. In Figure 1.2, each of three memories receives

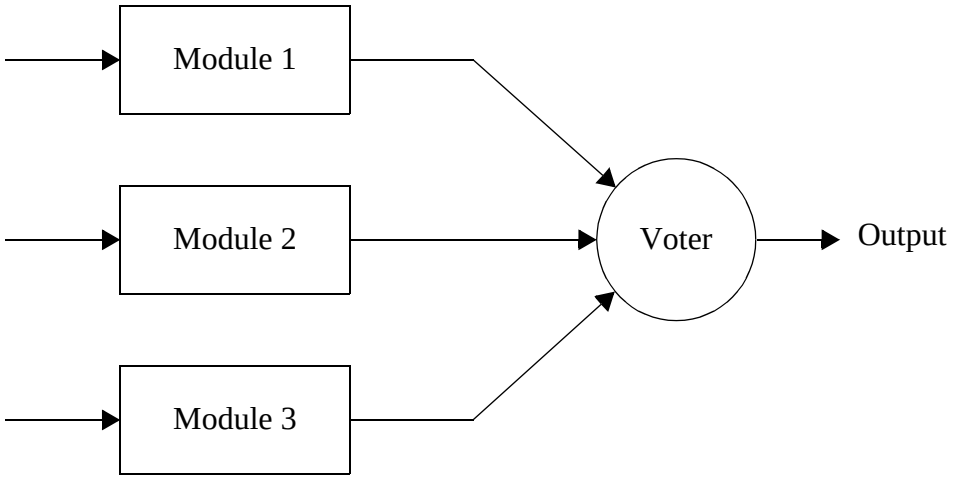



**Figure 1.1** Basic concept of Triple Modular Redundancy (TMR) [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 52].

data from a voter which has received its inputs from the three separate processors. If one processor fails, each memory will continue to receive a correct value because its voter will correct the corrupted value. A TMR system with triplicated voters is commonly called a *restoring organ* because the configuration will produce three correct outputs even if one input is faulty. In essence, the TMR with triplicated voters restores the error-free signal.

A generalization of the TMR approach is the N-modular redundancy (NMR) technique. NMR applies the same principle as TMR but uses N of a given module as opposed to only three. In most cases, N is selected as an odd number so that a majority voting arrangement can be used. The advantage of using N modules rather than three is that more module faults can often be tolerated. For example, a 5MR system contains five replicated modules and a voter. A majority voting arrangement allows the 5MR system to produce correct results in the face of as many as two module faults. In many critical-computation applications, two faults must be tolerated to allow the required reliability and fault tolerance capabilities to be achieved. The primary tradeoff in NMR is the fault tolerance achieved versus the hardware required. Clearly, there must be some limit in practical applications on the amount of redundancy that can be employed. Power, weight, cost, and size limitations very often determine the value of N in an NMR system.

Voting within NMR systems can occur at several points. For example, an industrial controller can sample the temperature of a chemical process from three independent sensors, perform a vote to determine which of the three sensor values to use, calculate the amount of heat or cooling to provide to the process (the calculations being performed by three or more separate modules), and then vote on the calculations to determine a result. The voting can be performed on both analog and digital data. The alternative, in this example, might be to sample the temperature from three independent sensors, perform the calculations, and then provide a single vote on the final result. The primary difference between the two approaches is fault containment. If voting is not performed on the temperature values from the sensors, then the effect of a sensor fault is allowed to propagate beyond the sensors and into the primary calculations. Voting at the sensors, however, will mask, and contain, the effects of a sensor fault. Providing several levels of voting, however, does require additional redundancy, and the benefits of fault containment must be compared to the cost of the extra redundancy.

In addition to there being a number of design tradeoffs on voting, there are several problems with the voting procedure, as well. The first is deciding whether a hardware voter will be used, or whether the voting process will be implemented in software. A software voter takes advantage of the computational capabilities available in a processor to perform the voting process with a minimum amount of additional hardware. Also, the software voter provides the ability to modify the manner in which the voting is performed by simply modifying the software. The disadvantage of the software voter is that the voting can require more time to perform simply because the processor cannot execute instructions and process data as rapidly as a dedicated hardware voter. The decision to use hardware or software voting will typically depend upon: (1) the availability of a processor to perform the voting, (2) the speed at which voting must be performed, (3) the criticality of space, power, and weight limitations, (4) the number of different voters that must be provided, and (5) the flexibility required of the voter with respect to future changes in the system. The concept of software voting is shown in Figure 1.3. Each processor exercutes its own version of task A. Upon completion of the tasks, each processor shares its results with processor 2, who then votes on the results before using them as input to task B. If necessary, each processor might also execute its version of

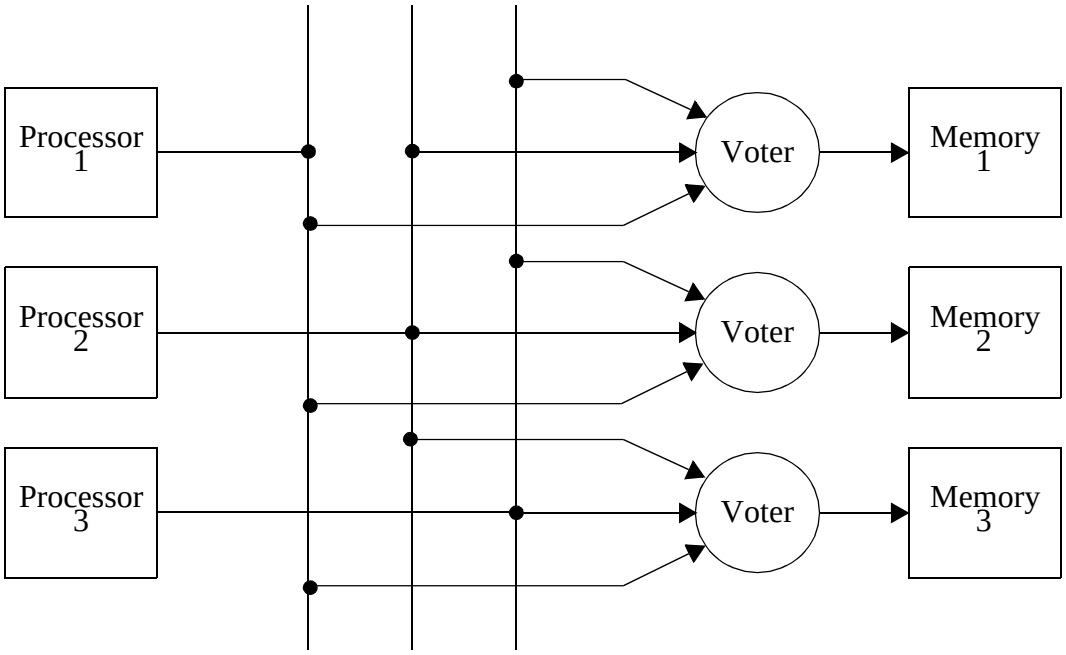



**Figure 1.2** The use of triplicated voters in a TMR configuration [Adapted from, Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 327].

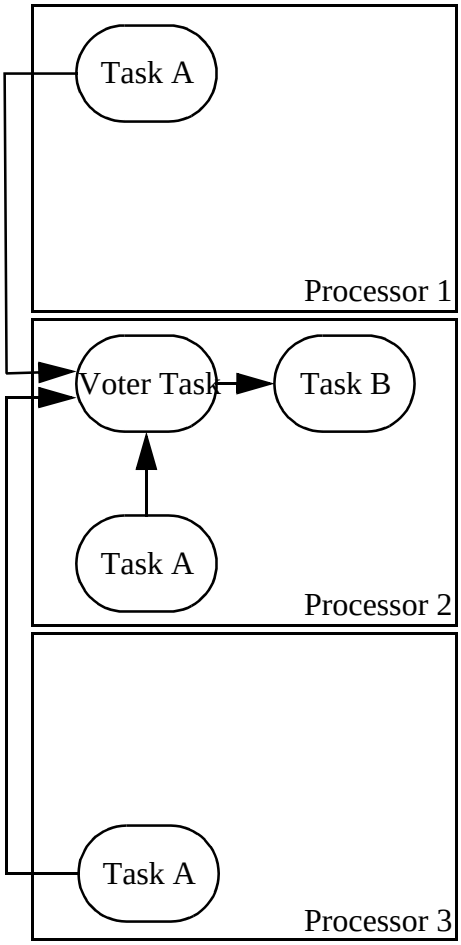



**Figure 1.3** Example of software voting.

the voting routine and receive data from the other processors.

A second major problem with the practical application of voting is that the three results in a TMR system, for example, may not completely agree, even in a fault-free environment. The sensors that are used in many control systems can seldom be manufactured such that their values agree exactly. Also, an analog-to-digital converter can produce quantities that disagree in the least-significant bits, even if the exact signal is passed through the same converter multiple times. When values that disagree slightly are processed, the disagreement can propagate into larger discrepancies. In other words, small differences in inputs can produce large differences in outputs that can significantly affect the voting process. Consequently, a majority voter may find that no two results agree exactly in a TMR system, even though the system may be functioning perfectly.

One approach that alleviates the problem of the previous paragraph is called the *mid-value select* technique. Basically, the mid-value select approach chooses a value from the three available in a TMR system by selecting the value that lies between the remaining two. As an example, consider Figure 1.4. If three signals are available, and two of those signals are uncorrupted and the third is corrupted, one of the uncorrupted results should lie between the other uncorrupted result and the corrupted result. The mid-value select technique can be applied to any system that uses an odd number of modules such that one signal must lie in the middle of the others.

The major difficulty with most techniques that use some form of voting is that a single result must ultimately be produced, thus creating a potential point where one failure can cause a system failure. Clearly, single-points-of-failure are to be avoided if a system is to be truly fault-tolerant. The need for a single result is apparent in many applications. For example, banking systems must

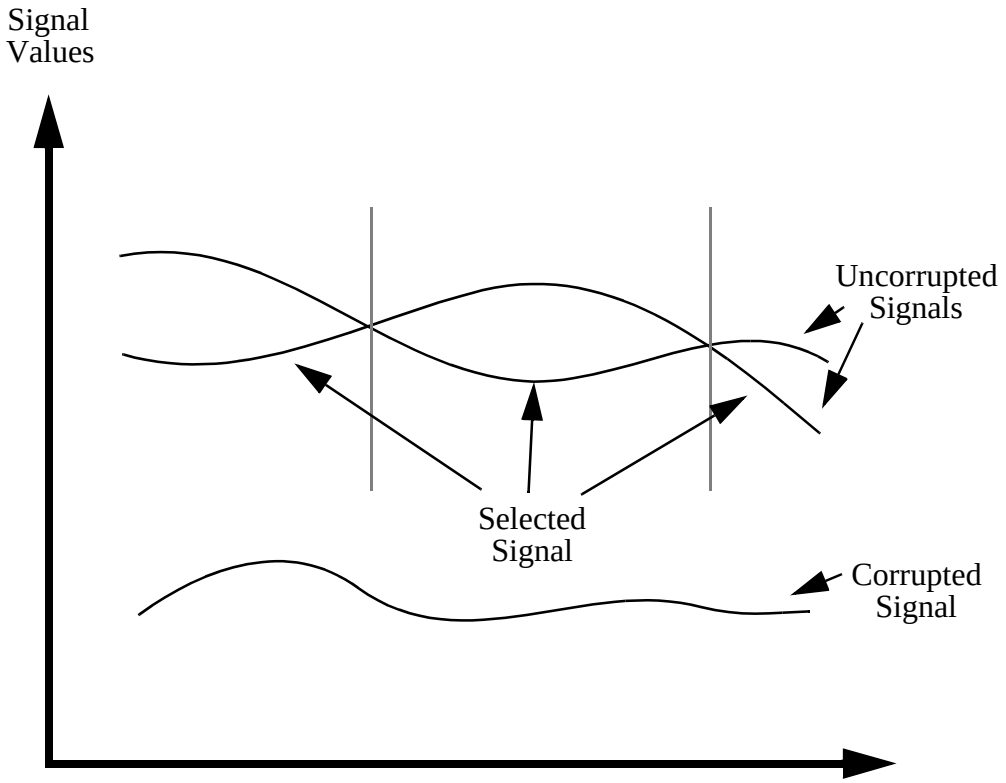



**Figure 1.4** Example illustration of the mid-value selection technique [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 60].

display one balance for each checking account, not three. Even though the banking computers may vote internally on some of the results, one result must ultimately be created. The same is true in critical-computation applications such as aircraft flight control. Most aircraft, even military aircraft, do not have redundancy of the actuators, or motors, that physically move the control surfaces. Consequently, a single control signal must be provided. There are several approaches that have been used successfully to create single results from redundant computations. We will consider the *flux-summing* technique.

The fundamental concept of flux-summing is illustrated in Figure 1.5. Here, a TMR system is employed to control the armature current of a small motor. The flux-summing approach uses the inherent properties of closed-loop control systems to compensate for faults. The flux-summer is a transformer that has three primary windings and a single secondary winding. The current produced in the secondary winding is proportional to the sum of the individual currents in the three primary windings. Under fault-free circumstances, each module provides approximately one-third of the total current produced in the secondary winding.

If a module fails, there are several scenarios that can result. First, the faulty module may stop providing current to the transformer. In this case, the motor will lose approximately one-third of the current necessary to maintain the present shaft position, or shaft velocity, depending on what quantity is being controlled. The remaining two modules will sense, via the feedback path, that the motor is deviating from the desired position or velocity. In other words, the error signal produced by each module as part of the closed-loop control process will increase in magnitude. The result will be that the two fault-free modules will change the current they are providing to offset the loss of current from the faulty module.

A second failure scenario that can result is that one module may fail so as to provide a maximum current to the flux-summer, regardless of the input signal values. The inherent feedback of the system will once again compensate for the condition by modifying the currents produced by the remaining fault-free modules. One way to visualize the response of the flux-summer is to consider one of the fault-free modules as providing a current of equal magnitude but opposite polarity of the faulty module so as to cancel the effect of the faulty module. The remaining fault-free mod-

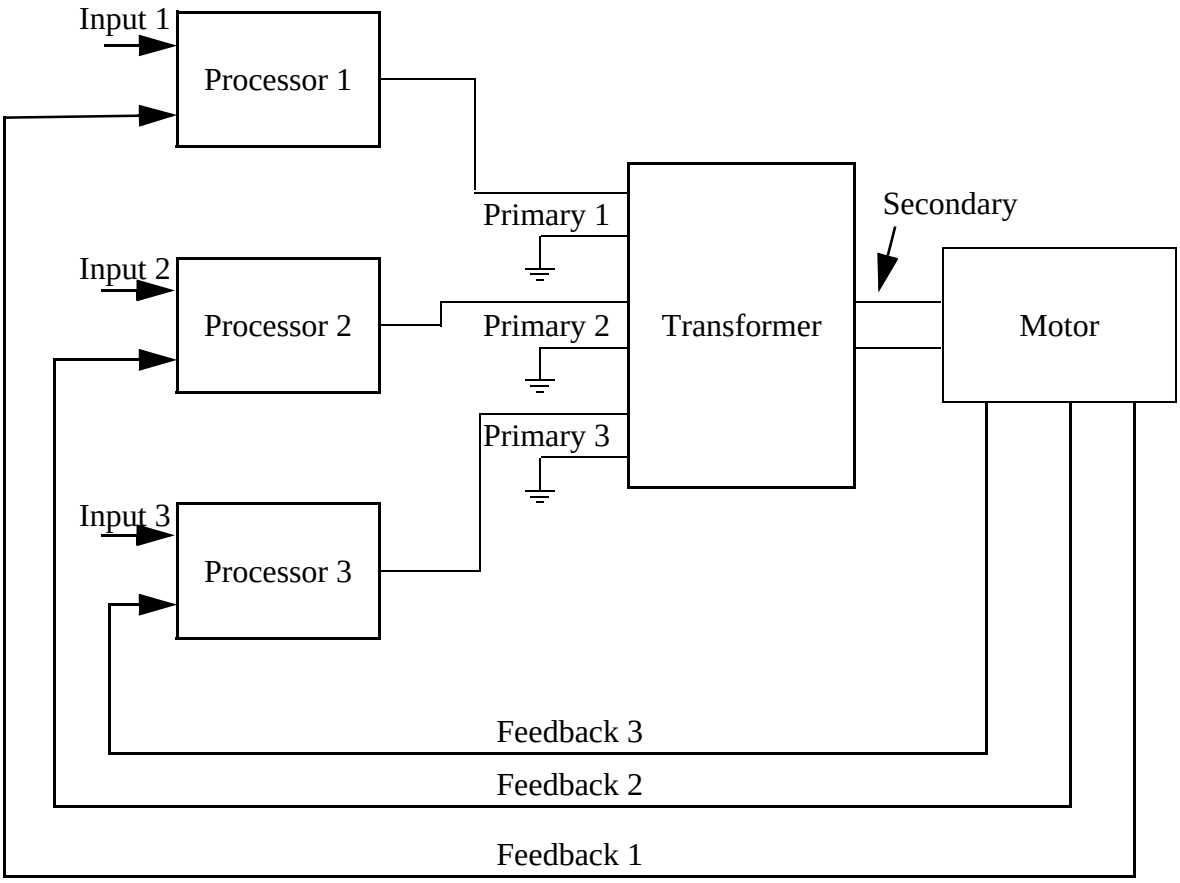



**Figure 1.5** Basic concept of flux-summing [Adapted from Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 61].

ule is then capable of controlling the system.

It is important to understand that the flux-summing approach is not a voting process, but it has the same effect of masking faults. The flux-summer can be used in the basic TMR approach or in the more general NMR technique. The primary limitation is the number of coils that can be physically mounted on an iron core. The flux-summers can be designed in a very reliable manner and are extremely insensitive to external disturbances of various types.

**Active Hardware Redundancy**. Active hardware redundancy techniques attempt to achieve fault tolerance by fault detection, fault location, and fault recovery. In many designs faults can be detected because of the errors they produce, so in many instances error detection, error location,

and error recovery are the appropriate terms to use. The property of fault masking, however, is not obtained in the active redundancy approach. In other words, there is no attempt to prevent faults from producing errors within the system. Consequently, active approaches are most common in applications where temporary, erroneous results are acceptable as long as the system reconfigures and regains its operational status in a satisfactory length of time. Satellite systems are good examples of applications of active redundancy. Typically, it is not catastrophic if satellites have infrequent, temporary failures. In fact, it is usually preferable to have temporary failures than to provide the large quantities of redundancy necessary to achieve fault masking.

The basic operation of an active approach to fault tolerance is shown in Figure 1.6. During the normal operation of a system a fault can obviously occur. After the fault latency period, the

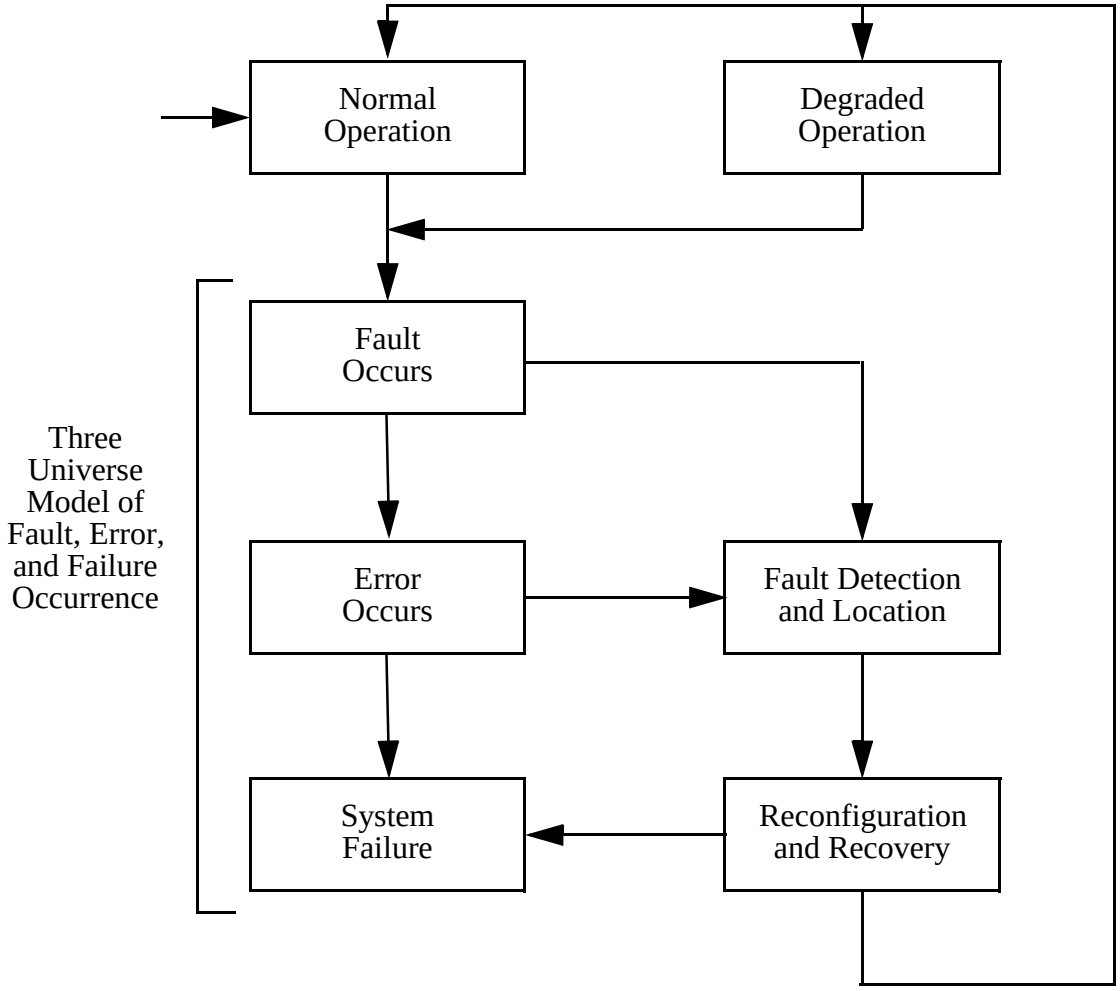



**Figure 1.6** Basic operation of an active approach to fault tolerance.

fault will produce an error which is either detected or it is not detected. If the error remains undetected, the result will be a system failure. The failure will occur after a latency period has expired. If the error is detected, the source of the error must be located, and the faulty component removed from operation. Next, a spare component must be enabled, and the system brought back to an operational state. It is important to note that the new operational state may be identical to the original operational state of the system or it may be a degraded mode of operation. The processes of fault location, fault containment, and fault recovery are normally referred to simply as reconfiguration. It is clear from this description that active approaches to fault tolerance require fault detection and location capabilities.

One example of a fault detection mechanism used in active redundancy is the simple *duplication with comparison* scheme. The basic concept of duplication with comparison is to develop two identical pieces of hardware, have them perform the same computations in parallel, and compare the results of those computations. In the event of a disagreement, an error message is generated. In its most basic form, the duplication concept cannot tolerate faults but can only detect them because there is no method for determining which of the two modules is faulty. However, duplication with comparison can be used as the fundamental fault detection technique in an active redundancy approach.

There are several potential problem areas in the duplication with comparison method of active redundancy. First, if the modules both receive the same input, then a failure of the input device, or the lines over which the input signals must be transmitted, will cause both modules to produce the same, erroneous results. Second, the comparator may not be able to perform an exact comparison, depending on the application area. While duplicated telephone switching processors may always exactly agree, if they are fault-free, the processors in a digital control application may never exactly agree. Finally, faults in the comparator can cause an error indication when no error exists, or, worse yet, the comparator can fail such that eventual faults in the duplicated modules are never detected.

A technique that can be used in a duplicated microprocessor system to overcome some of the problems mentioned in the previous paragraph is illustrated in Figure 1.7. The basic concept is to implement the comparison process in software that executes in each of the microprocessors. Each processor has its own private memory to store programs and data. In addition, there is a shared

memory that can be used to transfer results from one processor to the other for comparison purposes. If the shared memory fails, both processors will detect a disagreement between the data stored in their own memories and that contained in the shared memory. If one of the processors fails, the condition can also be detected using this approach.

The comparisons between the two processors can be performed in one of several ways. The first, and most straightforward technique, is to simply compare each digital word bit-by-bit. Bit-by-bit comparisons have the same problems as found in voting circuits. Specifically, many applications will require comparing digital words that do not agree exactly, even though the system is

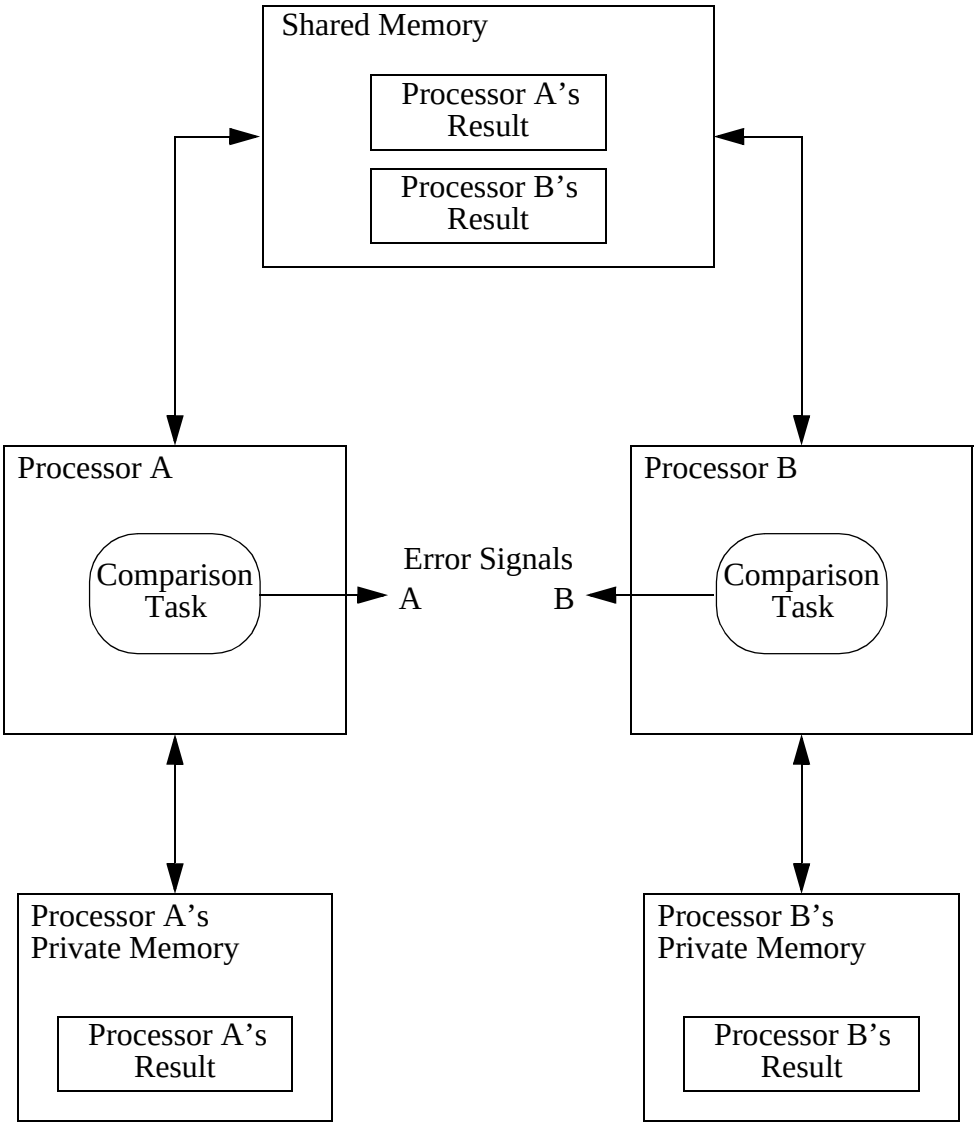



**Figure 1.7** A software implementation of duplication with comparison.

fault-free. In such cases, it is common to compare only the most-significant bits of words. For example, in the comparison of two n-bit words, the k least-significant bits might be ignored because their impact on the system is negligible. The feasibility of using such a comparison technique is clearly dependent upon the application. In banking systems, for example, bank balances calculated on duplicate processors must agree exactly. In many real-time control applications, however, minor differences are expected and have little, if any, impact.

A second form of active hardware redundancy, is called the *standby replacement* or *standby sparing* technique. In standby sparing, one module is operational while one or more modules serve as standbys, or spares. Various fault detection or error detection schemes are used to determine when a module has become faulty, and fault location is used to determine exactly which module, if any, is faulty. If a fault is detected and located, the faulty module is removed from operation and replaced with one of the spares. The reconfiguration operation in standby sparing can be viewed conceptually as a switch whose output is selected from one, and only one, of the modules providing inputs to the switch. The switch examines error reports from the error detection circuitry associated with each module to decide which module's output to use. If all modules are providing error-free results the selection can be made using a fixed priority. Any module that provides erroneous results is eliminated from consideration.

Standby sparing can bring a system back to full operational capability after the occurrence of a fault, but it requires that a momentary disruption in performance occur while the reconfiguration is performed. If the disruption in processing must be minimized, *hot standby sparing* can be used. In the hot standby sparing technique, the spares operate in synchrony with the on-line modules and are prepared to take over at any time. In contrast to hot standby sparing is *cold standby sparing* where the spares are unpowered until needed to replace a faulty module. The disadvantage of the cold sparing approach is the time required to apply power to a module and perform initialization prior to bringing the module into active service. The advantage of cold standby sparing is that spares do not consume power until needed to replace a faulty module. A satellite application where power consumption is extremely critical is an example where cold standby sparing may be desireable, or required. A process control system that controls a chemical reaction is an example where the reconfiguration time needs to be minimized, and cold standby sparing is undesireable, or unus-

able.

One variation on the standby sparing technique is called the *pair-and-a-spare* approach. The technique combines the features present in both standby sparing and duplication with comparison. In essence, the pair-and-a-spare approach uses standby sparing; however, two modules are operated in parallel at all times and their results are compared to provide the error detection capability required in the standby sparing approach. The error signal from the comparison is used to initiate the reconfiguration process that removes faulty modules and replaces them with spares. The reconfiguration process can be viewed conceptually as a switch that accepts the modules' outputs and error reports and provides the comparator with the outputs of two modules, one of which forms the output of the system. As long as the two selected outputs agree, the spares are not used. When a miscompare occurs, however, the switch uses the error reports from the modules to first identify the faulty module and then select a replacement module. In other words, the switch uses the error information from the comparator and the individual modules to maintain two fault-free modules operating in a duplication with comparison arrangement.

A variation on the pair-and-a-spare technique is to always operate modules in pairs. During the design, modules are permanently paired together, and when one module fails, neither module in the pair is used. In other words, modules are always operated and discarded in pairs so that the specific identification of which module is faulty is never required, only the identification of a faulty pair is necessary. Faulty pairs are easily identified based on the outcome of the comparison process.

**Hybrid Hardware Redundancy**. The fundamental concept of hybrid hardware redundancy is to combine the attractive features of both the active and the passive approaches. Fault masking is used to prevent the system from producing erroneous results, and fault detection, fault location, and fault recovery are used to reconfigure the system in the event of a fault. Hybrid redundancy is usually very expensive in terms of the amount of hardware required to implement a system; consequently, hybrid redundancy is most often used in applications that require extremely high integrity of the computations.

There are several approaches to hybrid redundancy. Most, however, are based upon the concept of *N-modular redundancy (NMR) with spares*. The idea of NMR with spares is to provide a

basic core of N modules arranged in a voting, or a form of voting, configuration. In addition, spares are provided to replace failed units in the NMR core. The benefit of NMR with spares is that a voting configuration can be restored after a fault has occurred. For example, a design that uses TMR with one spare will mask the first module fault that occurs. If the faulty module is then replaced with the spare unit, the second module fault can also be masked, thus providing tolerance of two module faults. For a passive approach to tolerate two module faults, five modules must be configured in a fault masking arrangement. The hybrid approach can accomplish the same results using only four modules and some fault detection, location, and recovery techniques.

The NMR with spares technique is illustrated in Figure 1.8. The system will remain in the basic NMR configuration until the disagreement detector determines that a faulty unit exists. One approach to fault detection is to compare the output of the voter with the individual outputs of the modules. A module that disagrees with the majority is labeled as faulty and removed from the NMR core. A spare unit is then switched in to replace the faulty module. The reliability of the basic NMR system is maintained as long as the pool of spares is not exhausted. Voting always occurs among the active participants in the NMR core, masking faults and ensuring continuous, error-free computations.

A variation on the basic hybrid redundancy technique is called the *triple-duplex* approach because it combines duplication with comparison and triple modular redundancy. The use of TMR allows faults to be masked and continuous, error-free performance to be provided for up to one faulty module. The use of the duplication with comparison allows faults to be detected and faulty modules removed from the TMR voting process and replaced with spares. The basic structure of the triple-duplex architecture is shown in Figure 1.9. To allow faults to be detected, each module is constructed using the duplication with comparison method. If the comparison process detects a fault, the faulty module is removed from the voting arrangement and replaced with a spare. The removal of faulty modules allows future faults to be tolerated.

The three primary hardware redundancy techniques -- passive, active, and hybrid -- each have advantages and disadvantages that are important in different applications. The key differences are: (1) passive techniques rely strictly on fault masking, (2) active techniques do not use fault masking

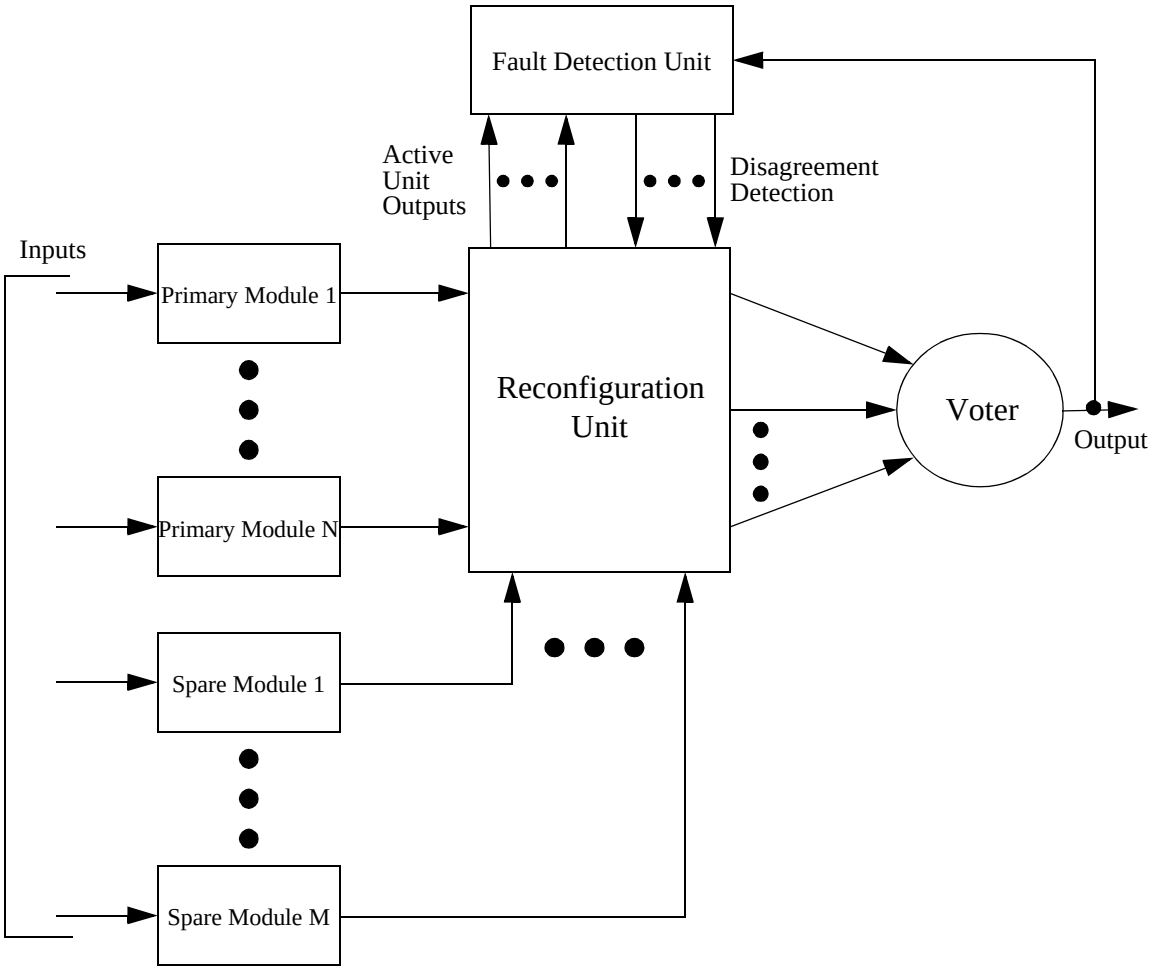



**Figure 1.8** Hybrid redundancy approach using NMR with spares [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 70].

but instead employ detection, location, and recovery techniques (reconfiguration), and (3) hybrid approaches employ both fault masking and reconfiguration.

The choice of a hardware approach depends heavily on the application. Critical-computation applications usually mandate some form of either passive or hybrid redundancy because momentary, erroneous results are not acceptable in such systems. The highest reliability is usually achieved using the hybrid techniques. In long-life and high-availability applications active approaches are often used because it is typically acceptable to have temporary, erroneous outputs; the important thing is that the system can be restored to an operational state in a short amount of

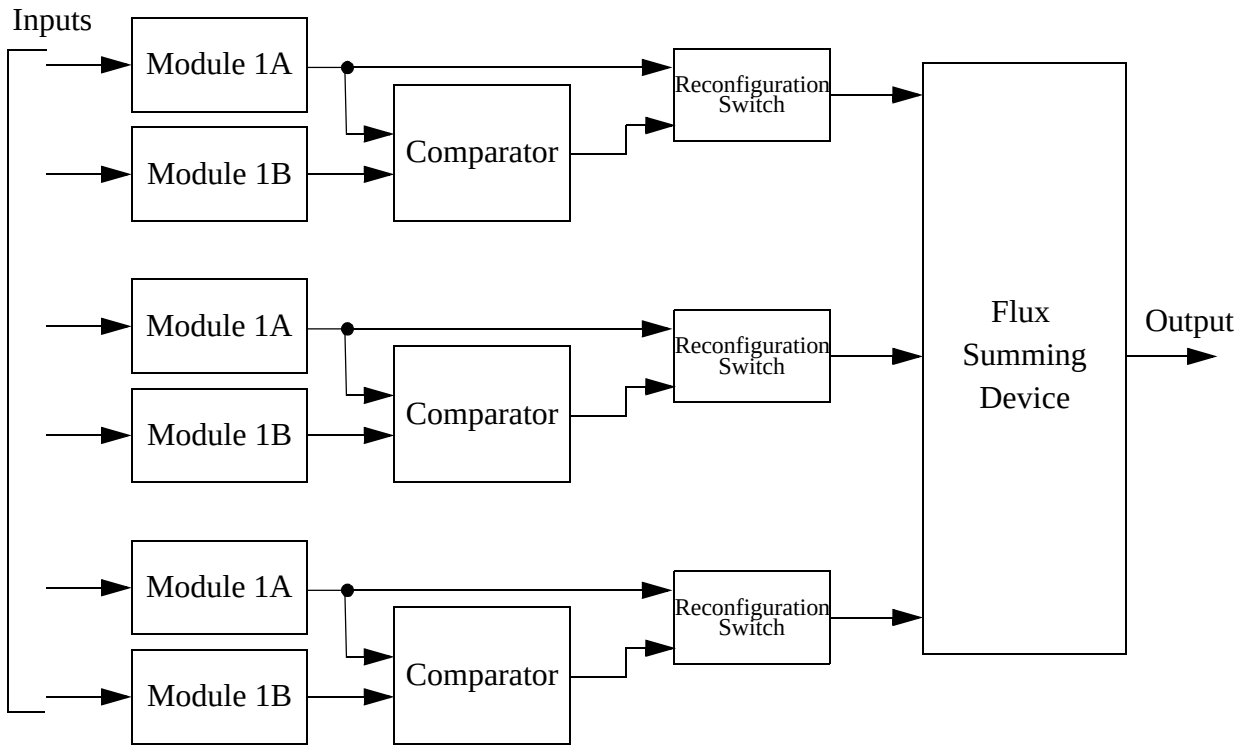



**Figure 1.9** The triple-duplex approach to hybrid redundancy [Adapted from Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 80].

time using reconfiguration techniques.

The cost, in terms of hardware, of the redundancy techniques increases as we go from active to passive and finally to hybrid. Active techniques typically use less hardware but have the disadvantage of potentially producing momentary, erroneous outputs. Passive techniques provide fault masking but use substantial investments in hardware. Finally, hybrid approaches provide the advantage of fault masking but require enough hardware to use voting and require hardware for spares. Hybrid approaches are typically the most costly in terms of hardware and are used when the highest levels of reliability are required.

## 1.2.2. Information Redundancy

Information redundancy is the addition of redundant information to data to allow fault detection,

fault masking, or possibly fault tolerance. Good examples of information redundancy are error detecting and error correcting codes, formed by the addition of redundant information to data words, or by the mapping of data words into new representations containing redundant information. Before beginning the discussions of various codes, we will define several basic terms that will appear throughout this section [TANG 69].

In general, a *code* is a means of representing information, or data, using a well-defined set of rules. A *code word* is a collection of symbols, often called digits if the symbols are numbers, used to represent a particular piece of data based upon a specified code. A *binary code* is one in which the symbols forming each code word consist of only the digits 0 and 1. For example, a Binary Coded Decimal (BCD) code defines a 4-bit code word for each decimal digit. The BCD code, for example, is clearly a binary code. A code word is said to be *valid* if the code word adheres to all of the rules that define the code; otherwise, the code word is said to be *invalid*.

The *encoding operation* is the process of determining the corresponding code word for a particular data item. In other words, the encoding process takes an original data item and represents it as a code word using the rules of the code. The *decoding operation* is the process of recovering the original data from the code word. In other words, the decoding process takes a code word and determines the data that it represents. Of primary interest here are binary codes. In many binary code words, a single error in one of the binary digits will cause the resulting code word to no longer be correct, but, at the same time, the code word will be valid. Consequently, the user of the information has no means of determining the correctness of the information. It is possible, however, to create a binary code for which the valid code words are a subset of the total number of possible combinations of 1s and 0s. If the code words are formed correctly, errors introduced into a code word will force it to lie in the range of illegal, or invalid code words, and the error can be detected. This is the basic concept of the *error detecting codes*. The basic concept of the *error correcting code* is that the code word is structured such that it is possible to determine the correct code word from the corrupted, or erroneous, code word. Typically, the code is described by the number of bit errors that can be corrected. For example, a code that can correct single-bit errors is called a *single-error correcting code*. A code that can correct two-bit errors is called a *double-error correcting code*, and so on.

A fundamental concept in the characterization of codes is the *Hamming distance*. The *Hamming distance* between any two binary words is the number of bit positions in which the two words differ. For example, the binary words 0000 and 0001 differ in only one position, and therefore have a Hamming distance of 1. The binary words 0000 and 0101, however, differ in two positions; consequently, their Hamming distance is 2. Clearly, if two words have a Hamming distance of 1, it is possible to change one word into the other simply by modifying one bit in one of the words. If, however, two words differ in two bit positions, it is impossible to transform one word into the other by changing one bit in one of the words.

The Hamming distance gives insight into the requirements of error detecting codes and error correcting codes. We define the *distance* of a code as the minimum Hamming distance between any two valid code words. If a binary code has a distance of two, then any single-bit error introduced into a code word will result in the erroneous word being an invalid code word because all valid code words differ in at least two bit positions. If a code has a distance of three, then any single-bit error or any double-bit error will result in the erroneous word being an invalid code word because all valid code words differ in at least three positions. However, a code distance of three allows any single-bit error to be corrected, if it is desired to do so, because the erroneous word with a single-bit error will be a Hamming distance of one from the correct code word and at least a Hamming distance of two from all others. Consequently, the correct code word can be identified from the corrupted code word.

A second fundamental concept of codes is *separability*. A *separable code* is one in which the original information is appended with new information to form the code word, thus allowing the decoding process to consist of simply removing the additional information and keeping the original data. In other words, the original data is obtained from the code word by stripping away extra bits, called the code bits or check bits, and retaining only those associated with the original information. A *nonseparable code* does not possess the property of separability, and, consequently, requires more complicated decoding procedures.

**Parity Codes.** Perhaps the simplest form of a code is the parity code. The basic concept of parity is very straightforward, but there are variations on the fundamental idea. Single-bit parity codes require the addition of an extra bit to a binary word such that the resulting code word has

either an even number of 1s or an odd number of 1s. If the extra bit results in the total number of 1s in the code word being odd, the code is referred to as *odd parity*. If the resulting number of 1s in the code word is even, the code is called *even parity*. If a code word with odd parity experiences a change in one of its bits, the parity will become even. Likewise, if a code word with even parity encounters a single-bit change, the parity will become odd. Consequently, a single-bit error can be detected by checking the number of 1s in the code word. The single-bit parity code (either odd or even) has a distance of two, therefore allowing any single-bit error to be detected but not corrected. It is important to note that the basic parity code is a separable code.

The most common application of parity is in the memories of computer systems. Before being written to memory, a data word is encoded to achieve the correct parity; the encoding consists of appending a bit to force the resulting word to have the appropriate number of 1s. When the data word is subsequently read from memory, the parity is checked to verify that it has not changed as a result of a fault within the memory. If an error is detected, the user of the memory is notified via an error signal that a potential problem exists. The extra information (the extra bit appended to each word) requires additional hardware to handle. For example, the memory must contain one extra bit per word to store the extra information, and the hardware must be designed to create and check the parity bit. As can be seen, the information redundancy concept often requires hardware redundancy as well.

One of the biggest problems with single-bit parity codes is their inability to guarantee the detection of some very common multiple-bit errors. For example, a memory can be constructed from individual chips that each contain several bits; four bits is a very common number. If a chip in the memory becomes faulty, the simple parity code may be unable to detect the resulting error because multiple bits are affected. The basic parity scheme can be modified to provide additional error detection capability. There are five basic parity approaches including the fundamental odd and even parity; bit-per-word, bit-per-byte, bit-per-chip, bit-per-multiple-chips, and interlaced parity.

The *bit-per-word* parity concept has already been discussed. The basic idea is to append one parity bit to each word. The primary disadvantage of the bit-per-word approach is that certain errors can go undetected. For example, if a word, including the parity bit, becomes all 1s because

of a complete failure of a bus or a set of data buffers, the odd parity method will only detect the condition if the total number of bits, including the parity bit, is even. Likewise, even parity will only detect this problem if the total number of bits is odd. In a similar manner, the condition of all bits becoming 0 will never be detected by the even bit-per-word parity method because 0 is considered to be an even number of 1s. Odd bit-per-word parity will always detect the condition of all bits being 0.

An alternate approach that uses parity is the *bit-per-byte* technique. Here, two parity bits are used on two separate portions of the original data. The technique is called bit-per-byte, but the parity groups can be any number of bits; not just the eight bits normally associated with the terminology, byte. To gain the full advantages of the approach, however, the number of information bits associated with each parity bit should be even. Also, the parity of one group should be even while the parity of the other group should be odd. The primary advantage of this approach is the ability to detect both the *all 1s* and *all 0s* conditions. If the complete code word becomes all 1s, the even parity bit will be erroneous. If the complete code word becomes all 0s, the odd parity bit will be erroneous. In both cases, the erroneous conditions are detected. The bit-per-byte technique also provides additional protection against multiple-bit errors; for example, two-bit errors will always be detected as long as one bit is in the even parity group and the other is in the odd parity group.

The fundamental disadvantage of both the bit-per-word and bit-per-byte parity approaches is the ineffective detection capability for multiple-bit errors. Many memories are organized using memory chips that contain either four bits, eight bits, or more, of memory. Several of these chips are then used in parallel to form the complete number of bits of each word in the memory. If one chip fails (this is called the *whole-chip failure mode*), several bits of each word of memory can be affected. Therefore, the single-bit error assumption is often ineffective.

One approach that is useful to detect the failure of a complete chip is the *bit-per-multiple-chips* method. The basic concept is to have one bit from each chip of the memory associated with a single parity bit. Sufficient parity bits are provided to allow each data bit within a chip to be associated with a distinct parity bit. For example, consider a 16-bit word which includes data bits 0 through 15. Suppose parity bit $P_0$ establishes the parity of a group of bits including bits 0, 4, 8, and 12; parity bit $P_1$ establishes the parity of a group including bits 1, 5, 9, and 13; parity bit $P_2$ estab-

lishes parity for the group 2, 6, 10, and 14; and $P_3$ establishes parity for the group 3, 7, 11, and 15. Assume that chip 0 contains all of the parity bits, chip 1 contains bits 0, 1, 2, and 3, chip 2 contains 4, 5, 6, and 7, chip 3 contains 8, 9, 10, and 11, and chip 4 contains 12, 13, 14, and 15. Notice that each parity group includes one, and only one, bit from each chip. If one chip fails, all of the parity groups will be affected, but no more than one bit in each parity group will be corrupted, so the parity code will detect the error.

One disadvantage of the bit-per-multiple-chip parity approach is that the failure of a complete chip is detected, but it is not located. The failure of any one of the chips will cause all parity groups to be in error, so the cause of the problem cannot be identified. One procedure that overcomes this problem is the *bit-per-chip* parity organization. Here, each parity bit is associated with one chip of the memory. Using the same 16-bit example as before, parity bit $P_0$ establishes correct parity for the group containing data bits 0, 1, 2, and 3. The other parity groups are defined accordingly. If a single bit becomes erroneous, the existence of the error is detected, and the chip that contains the erroneous bit is identified. This is extremely valuable from a maintenance standpoint; not only does the system have the capability to warn of the occurrence of a problem, but the system can direct the maintenance personnel to the source of the problem. The primary disadvantage of the bit-per-chip parity method is the susceptibility to the whole-chip failure mode. Because the basic parity code can detect only single errors, the multiple error condition associated with the failure of a complete chip can go undetected.

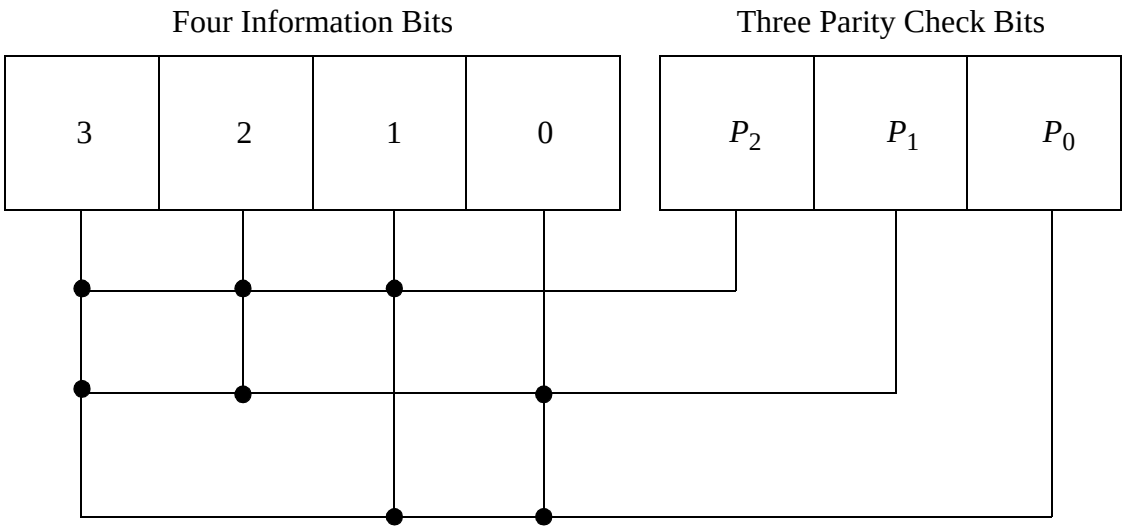
An alternate organization of the parity code is called *interlaced parity*. Interlaced parity is very similar in form to the bit-per-multiple-chips approach with one key difference. In interlaced parity, the parity groups are formed without regard to the physical organization of the memory. This is in contrast to the bit-per-multiple-chip organization which is intimately tied to the physical structure of the memory. In interlaced parity, the information bits are divided into equal-sized groups, and one parity bit is associated with each group. The bits of each group are then positioned such that no two adjacent bits are from the same parity group. This is accomplished by placing the first bit from group 0 in the least-significant bit position, the first bit from group 1 in the next most-significant position, the first bit from group 2 in the next position, and so on. Once the first bits of each group are placed, the remaining bits are added to the word in a similar manner. The interlaced

parity method is most often used when errors in adjacent bits are of major concern. Because no two adjacent bits are in the same parity group, errors in any two adjacent bits will be detected. A good example is a parallel bus; in many buses, two adjacent bits can become shorted together. The interlaced organization of parity will detect errors due to this type of fault.

The final organization of parity that will be considered is *overlapping parity*. In the overlapping parity approach, parity groups are formed with each bit appearing in more than one parity group. This is in contrast to the other approaches we have considered where each bit was contained in one and only one parity group. The primary advantage of overlapping parity is that errors can be located in addition to being detected. Once the erroneous bit is located, it can be corrected by a simple complementation, if desired. Overlapping parity is the basic concept of some of the Hamming error correcting codes.

Figure 1.10 illustrates the basic idea of overlapping parity when applied to four bits of information. Three parity groups are required to uniquely identify each erroneous bit in the four bits of information. The basic concept of overlapping parity is to place each bit in a unique combination of the parity groups. For example, referring to Figure 1.10, bit 3 appears in each parity group while bits 0, 1, and 2 appear in different combinations of two groups. If any one bit becomes erroneous, the impact is unique, as is illustrated in Figure 1.10. For example, if bit 3 is in error, all of the parity groups will be affected, but if bit 1 is erroneous, the parity groups associated with $P_0$ and $P_2$ will be affected, while $P_1$ will be unaffected. As shown in Figure 1.10, each possible single-bit error produces a unique impact on the parity of the three groups.

The overlapping parity approach can be implemented as an error correction scheme by using several comparators and a decoder in conjunction with the parity checking circuits. In addition, the correction process is performed by complementing the corrupted bit. For example, consider the case of four bits of information and three parity groups. When the four bits of information are written to memory, the three parity bits are generated and stored with the original four-bits of information as a single code word. When the code word is subsequently read from memory, the parity bits are regenerated using parity generation circuits. The regenerated parity bits are compared to the parity bits that were stored with the information in memory. The results of the comparisons can be




| Bit in Error | Parity Group Affected |
|---|---|
| 3 | $P_2$ $P_1$ $P_0$ |
| 2 | $P_2$ $P_1$ |
| 1 | $P_2$ $P_0$ |
| 0 | $P_1$ $P_0$ |
| $P_2$ | $P_2$ |
| $P_1$ | $P_1$ |
| $P_0$ | $P_0$ |

**Figure 1.10** The Fundamental Concept of Overlapped Parity [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 91].

used to uniquely identify which bit is corrupted, as is clear from Figure 1.10.

The penalty for using overlapping parity on four bits of information is high; three parity bits are required to detect and locate errors for the four bits of information, a redundancy of 75%. As the number of information bits increases, however, the number of parity bits required becomes a smaller percentage of the number of actual information bits. One can determine the required relationship between the number of information bits and the number of parity bits in a fairly simple manner. Let *m* be the number of information bits to be protected using an overlapping parity approach, and let *k* be the number of parity bits required to protect those *m* information bits. Each

bit error that can occur must produce a unique result when the parity is checked. With $k$ parity bits, there are $2^k$ unique outcomes of the parity checking process. With $k$ parity bits and $m$ information bits, there are $m+k$ different, single errors that can occur. So, we know that $2^k$ must be at least as large as $m+k$. Also, we must have a unique result of the parity checks when there is no error, so the total number of unique combinations must be at least as large as $m+k+1$. Therefore, the relationship between $k$ and $m$ must be

$$2^k \geq m + k + 1$$

**m-of-n Codes**. The basic concept of the *m-of-n* code is to define code words that are $n$ bits in length and contain exactly $m$ 1s. As a result, any single-bit error will force the resulting erroneous word to have either $m+1$ or $m-1$ 1s, and the error can be detected. The primary advantage of the *m-of-n* code is the conceptual simplicity; it is very easy to visualize the error detection process. The major disadvantage, however, is that the encoding, decoding, and the detection processes are often extremely difficult to perform, despite their conceptual simplicity.

The easiest way to construct an *m-of-n* code is to take the original $i$ bits of information and append $i$ bits. The appended bits are chosen such that the resulting $2i$-bit code words each have exactly $i$ 1s, therefore producing an *i-of-2i* code. The obvious disadvantage of using the *i-of-2i* code is that twice as many bits are required to represent the information; consequently, the redundancy of the code is 100%. The advantage of creating an *i-of-2i* code is that both the encoding and the decoding processes are simple because the code is separable. The encoding procedure can be performed by counting the number of 1s in the information to be encoded and looking up in a table the desired bits to append, based on the number of 1s in the original information. The decoding can be performed by simply removing the appended bits from the code word and retaining the original information.

It is easy to see that *m-of-n* codes have a distance of two. Any single-bit error in an *m-of-n* code word will change the number of 1s to either $m+1$ or $m-1$, depending upon whether the error changed a 0 to a 1 or a 1 to a 0. A second bit-error, however, can change the number of 1s back to $m$. For example, if one bit is changed from 0 to 1 and a second bit is changed from 1 to 0, the number of 1s in the code word remains unchanged. Consequently, the error goes undetected. If the

errors are all *unidirectional*, meaning that all errors are either a change of a 1 to a 0 or a change of a 0 to a 1, but not combinations of the two changes, the *m-of-n* code will provide detection of the multiple errors. Consequently, the *m-of-n* code provides detection of all single errors and all multiple, unidirectional errors. The *m-of-n* codes can often be constructed in a more efficient manner, but the separable nature of the code is usually lost.

**Duplication Codes.** *Duplication codes* are based on the concept of completely duplicating the original information to form the code word. The primary advantage of the duplication code is simplicity. The major disadvantage is clearly the number of extra bits that must be provided to allow the code to be constructed. Duplication codes are found in many applications, including memory systems and some communication systems. The encoding process for the duplication code consists of simply appending the original $i$ bits of information to itself to form a code word of length $2i$ bits. If a single-bit error occurs, the two halves of the code word will disagree, and the error can be detected. In communication systems, the duplication concept is often applied by transmitting all information twice; if both copies agree, the information is assumed to be correct. The penalty paid in the communications application is a decrease in the information rate of the system because $2i$ bits must be transmitted to obtain $i$ bits of information.

A variation on the basic duplication code is to complement the duplicated portion of the code word. The use of complemented duplication is particularly advantageous when the original information and its duplicate must be processed by the same hardware. The primary advantage of all variations of the duplication codes is the simplicity associated with generating the code words and the ease of obtaining the original information from the code word. The advantage is usually offset, however, by the cost of completely duplicating the original information. Also, it is usually very time consuming to implement the duplication codes. In memory applications, each word must be written and read twice. In communication applications, each word must be transmitted twice. In both cases, the time required to perform the operation is doubled. So, the duplication code often requires not only the 100% redundancy in information, but typically a 100% redundancy in hardware and time as well.

**Checksums.** The checksum is another form of separable code that is most applicable when blocks of data are to be transferred from one point to another. Examples where checksums are used

frequently include data transfers between mass storage devices -- such as disks -- and a computer, and packet-switched networks. The *checksum* is a quantity of information that is added to the block of data to achieve error detection capability. There are four primary types of checksums that are typically used; single-precision, double-precision, Honeywell, and the residue checksum.

The basic concept of the checksum is illustrated in Figure 1.11. When the original data is created, an additional piece of information, called the checksum, is appended to the block of data. The checksum is then regenerated when the data is received at the destination or, in some applications, when the data is read from memory. The regenerated checksum and the original checksum are compared to determine if an error has occurred in the data, the checksum generation, checksum regeneration, or the checksum comparison.

The checksum is basically the sum of the original data. The difference between the various forms of the checksum is the way in which the summation is generated. The simplest form of the checksum is the *single-precision checksum*. The *single-precision checksum* is formed by performing the binary addition of the data that is to be protected and ignoring any overflow that occurs. For example, if each data word is $n$ bits, the checksum will be $n$ bits as well. If the true binary sum of the data exceeds $2^n$-1, then an overflow will have occurred; in the single-precision checksum the overflow is ignored. In other words, the single-precision checksum is formed by adding the $n$-bit data in a modulo-$2^n$ fashion.

The primary difficulty with the single-precision checksum is that information, and as a result, error detection capability is lost in the ignored overflow. One technique that is often used to overcome the limitations of the single-precision checksum is to compute the checksum in double precision; thus, the name *double-precision checksum*. The basic concept of the double-precision checksum is to compute a $2n$-bit checksum for a block of $n$-bit words using modulo-$2^{2n}$ arithmetic. Overflow is still a concern, however, but it is now overflow from a $2n$-bit sum as opposed to an $n$-bit sum.

A third form of the checksum is called the *Honeywell checksum*. The basic idea of the *Honeywell checksum* is to concatenate consecutive words to form a collection of double-length words. For example, if there are $k$ $n$-bit words, a set of $\frac{k}{2}$ $2n$-bit words is formed, and a checksum is cal-

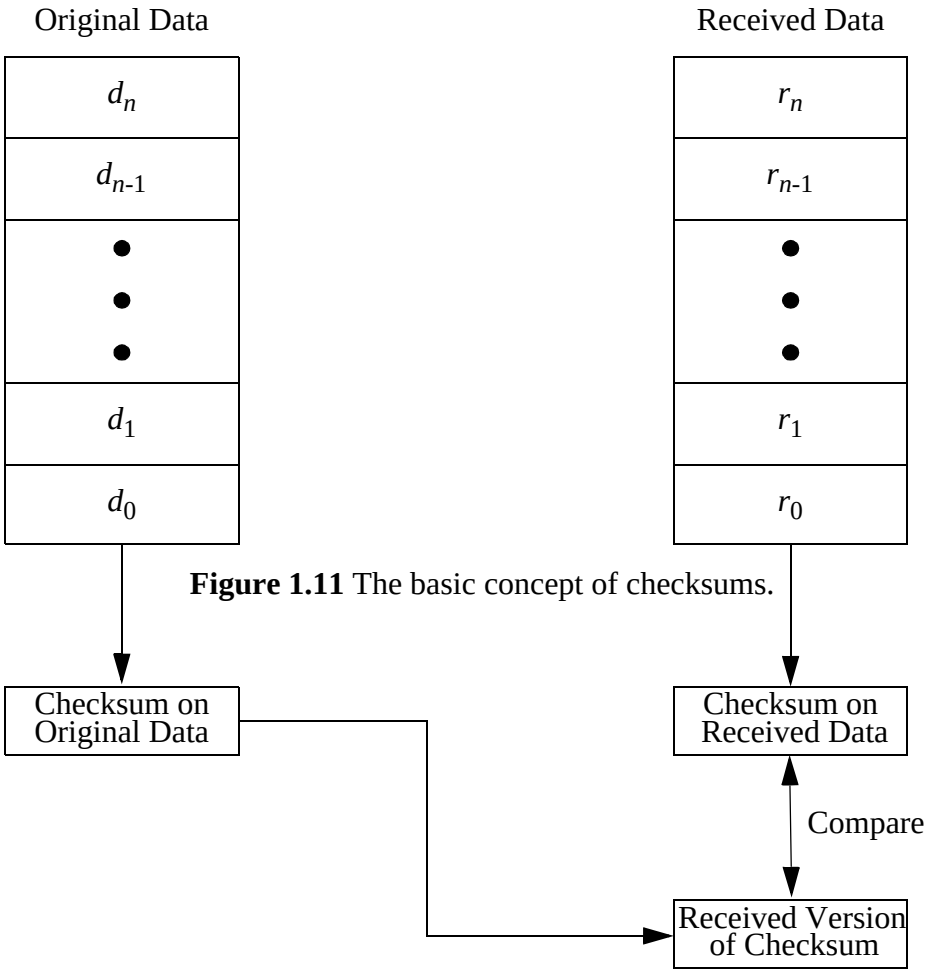



**Figure 1.11** The basic concept of checksums.

**Figure 1.11** The Basic Concept of Checksums [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 98].

culated over the newly-structured data. The primary advantage of the Honeywell checksum is that a bit error that appears in the same bit position of all words will affect at least two bit positions of the checksum. For example, if a complete column of the original data is erroneous, the modified data structure has two erroneous columns.

The final form of the checksum that will be considered is the *residue checksum*. The concept of the *residue checksum* is the same as the single-precision checksum except that the carry bit out of the most-significant bit position is not ignored but is added back to the checksum in an end-around carry fashion.

One important point concerning checksums is that they are capable of error detection but not

error location. If the checksum generated at the receiving point differs from the checksum generated at the transmission point, an error is indicated, but there is not enough information available to determine where the error has occurred. The complete block of data over which the checksum was formed must be corrected.

**Cyclic Codes.** The fundamental feature of *cyclic codes* is that any end-around shift of a code word will produce another code word [LIN 83]. In other words, the cyclic code is invariant to the end-around shift operation. Cyclic codes are frequently applied to sequential-access devices such as tapes, bubble memories, and disks. In addition, cyclic codes are extremely popular for use in data links. One reason the cyclic codes are attractive is because the encoding operation can be implemented using simple shift registers with feedback connections.

A cyclic code is characterized by its generator polynomial, *G(X)*, which is a polynomial of degree (*n*-*k*) or greater, where *n* is the number of bits contained in the complete code word produced by *G(X)*, and *k* is the number of bits in the original information to be encoded. For binary cyclic codes, the coefficients of the generator polynomial are all either 0 or 1. The integers *n* and *k* specify the characteristics of the cyclic code. A cyclic code with a generator polynomial of degree (*n*-*k*) is called an (*n*,*k*) cyclic code. Such codes possess the property of being able to detect all single errors and all multiple, adjacent errors affecting less than (*n*-*k*) bits [LIN 83]. The error detection property of cyclic codes is particularly important in communications applications where *burst errors* can occur. A *burst error* is the result of a transient fault and usually introduces a number of adjacent errors into a given data item. For example, a word that is transmitted serially can have several adjacent bits corrupted by a single disturbance; one would hope that the coding scheme could detect such errors. (*n*,*k*) cyclic codes will detect adjacent errors as long as the number of adjacent bits affected does not exceed (*n*-*k*).

The properties of cyclic codes are generated by representing the code words as coefficients of a polynomial. For example, suppose we have the code word, $v=(v_0, v_1, \ldots, v_{n-1})$. This code word corresponds to the polynomial *V(X)* where

$$V(X) = v_0 + v_1 X + v_2 X^2 + \ldots + v_{n-1} X^{n-1}$$

Each *n*-bit code word is represented by a polynomial of degree (*n*-1) or less. The polynomial, *V(X)*,

is called the *code polynomial* of the code word, *v*.

The code polynomials for a nonseparable cyclic code are generated by multiplying a polynomial, representing the data to be encoded, by another polynomial known as the generator polynomial. The generator polynomial determines the characteristics of the cyclic code. Any additions required during the multiplication of the two polynomials are performed using modulo-2 addition. For example, suppose that we have a generator polynomial, $G(X)= 1 + X + X^3$, and we wish to encode the binary data, (1101). The data, (1101), can be represented by the data polynomial, $D(X)= 1 + X + X^3$. The code polynomial is generated by multiplying the data polynomial and the generator polynomial. Specifically, the code polynomial is generated as $V(X)=D(X)G(X)=(1 + X + X^3)(1 + X + X^3)=1 + X^2 + X^6$. In more exact terms, the code polynomial is given by $V(X)=1 + (0)X + (1)X^2 + (0)X^3 + (0)X^4 + (0)X^5 + (1)X^6$, and the code word, *v*, consists of the coefficients of that code polynomial. In other words, *v*=(1010001).

Perhaps the most interesting aspect of the nonseparable cyclic codes is the manner in which they can be generated. Recall that the code polynomial is generated by multiplying the data polynomial by the generator polynomial and adding the coefficients in a modulo-2 fashion. If we consider the blocks labeled *X* as multipliers by the factor, *X*, and the addition elements as modulo-2 adders, the circuit shown in Figure 1.12 performs the multiplication of two polynomials. For example, if $D(X)=1$, the output, $V(X)$, of the circuit will be $1+X^2+X^3$. Likewise, if $D(X)=1+X+X^3$, the output will be given by

$$V(X) = 1 + X + X^3 + [X(1 + X + X^3) + (1 + X + X^3)]X^2$$

or

$$V(X) = 1 + X + X^2 + X^3 + X^4 + X^5 + X^6$$

Therefore, if the generator polynomial is $G(X)=1+X^2+X^3$, the circuit shown in Figure 1.12 will generate the code polynomial by multiplying the data polynomial by the generator polynomial. For example, if the data to be encoded is (1101), the data polynomial will be $D(X)=1+X+X^3$. The resulting code polynomial yields the code word (1111111).

The version of the cyclic code that has been presented thus far is not a separable code, so the

decoding process involves more than simply picking certain bits from the code word. The structure of the cyclic code, however, makes the decoding process relatively easy.

Suppose that we have a code word ($r_0, r_1, r_2, \ldots, r_{n-1}$), and we wish to determine if this code word is valid. We know that the code word, ($r_0, r_1, r_2, \ldots, r_{n-1}$), can be represented by the code polynomial, $R(X) = r_0 + r_1X + r_2X^2 + \ldots + r_{n-1}X^{n-1}$. We also know that the correct code polynomial was generated by multiplying the original data polynomial by the generator polynomial. In other words, if *R(X)* is a valid code polynomial, then it was generated as *R(X) = D(X)G(X)* where *G(X)* is the generator polynomial and *D(X)* is the original data polynomial. If we write

$$R(X) = D(X)G(X) + S(X)$$

then the quantity *S(X)* should be zero if the polynomial *R(X)* is a valid code polynomial. In other words, *R(X)* should be an exact multiple of the generator polynomial. One way to determine if *R(X)*

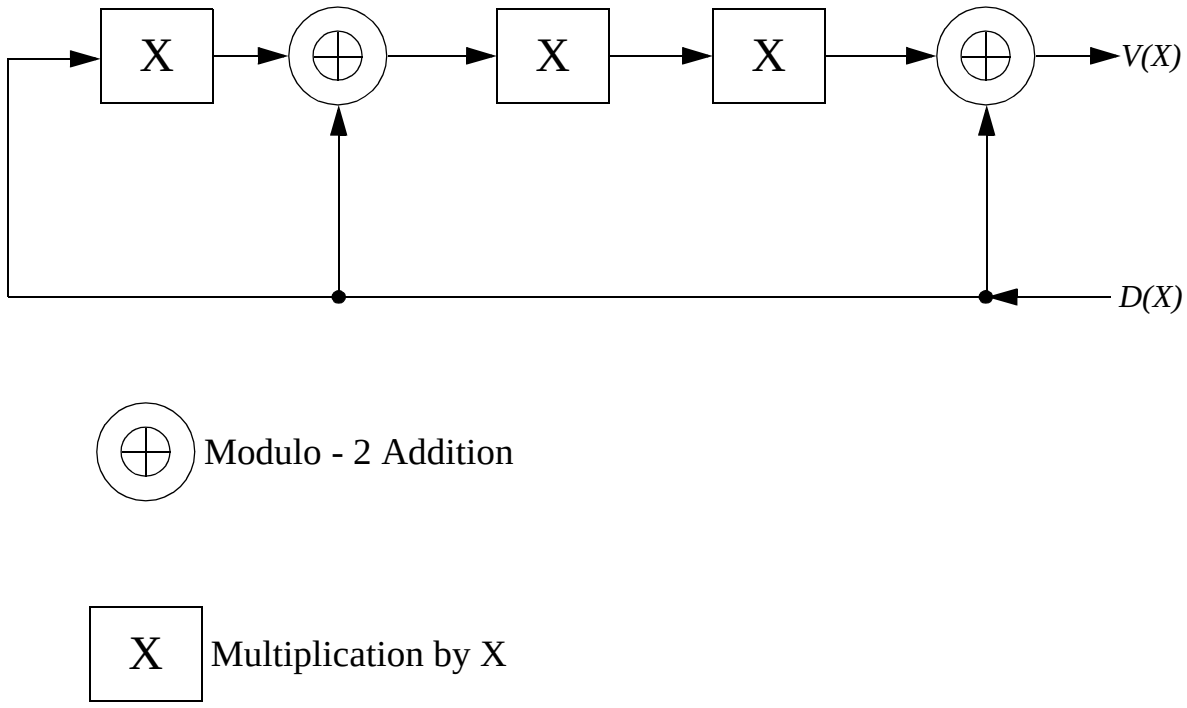



**Figure 1.12** Example Circuit for Generating a Cyclic Code [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 98].

is indeed an exact multiple of the generator polynomial is to divide the polynomial *R(X)* by the generator polynomial, *G(X)*, and check to see if the remainder of the division is zero. If the remainder is zero, the polynomial is an exact multiple of the generator polynomial and is a valid code polynomial. The quantity *S(X)* is called the *syndrome polynomial*.

The process of division may at first seem complicated and difficult to implement. It turns out to be quite simple, however, when feedback circuits similar to the cyclic code generators are used. The circuit shown in Figure 1.13, for example, is capable of dividing a polynomial by the polynomial $1 + X + X^3$. Once again, the blocks labeled as $X$ perform multiplication by the factor, $X$. The adders in the circuit of Figure 1.13 are modulo-2 adders. The polynomial that appears on line *B(X)* of the circuit will be given by

$$B(X) = (X^3 + X)D(X)$$

But the values present on line *D(X)* are determined by both line *B(X)* and line *V(X)*. Specifically,

$$V(X) + B(X) = D(X)$$

or

$$V(X) = D(X) - B(X) = D(X) - (X^3 + X)D(X)$$

Because the functions of addition and subtraction are the same in the modulo-2 system, we obtain

$$V(X) = (X^3 + X + 1)D(X)$$

or

$$D(X) = \frac{V(X)}{X^3 + X + 1}$$

The circuit of Figure 1.13 will divide the polynomial, *V(X)*, by the polynomial, $X^3 + X + 1$.

The primary disadvantage of the cyclic codes discussed thus far is that they are not separable. It is possible, however, to generate a separable, cyclic code [NELSON 86]. To generate an $(n,k)$ code, the original data polynomial, *D(X)*, is first multiplied by $X^{n-k}$, and the result is divided by the generator polynomial, *G(X)*, to obtain a remainder, *R(X)*. The code polynomial is then computed

as $V(X) = R(X) + X^{n-k}D(X)$, and $V(X)$ is an exact multiple of the generator polynomial, $G(X)$. Notice that the multiplication by $X^{n-k}$ can be performed by simply shifting the coefficients of the data polynomial. Also notice that the addition of the remainder polynomial is equivalent to simply appending the remainder to the polynomial $X^{n-k}D(X)$.

It is relatively straightforward to see the validity of the encoding process described in the previous paragraph. Suppose we have an arbitrary data polynomial, $D(X)$, and a generator polynomial, $G(X)$. The code polynomial, $V(X)$, is given by

$$V(X) = X^{n-k}D(X) + R(X)$$

where $R(X)$ is the remainder obtained when $X^{n-k}D(X)$ is divided by $G(X)$. In other words,

$$\frac{X^{n-k}D(X)}{G(X)} = Q(X) + \frac{R(X)}{G(X)}$$

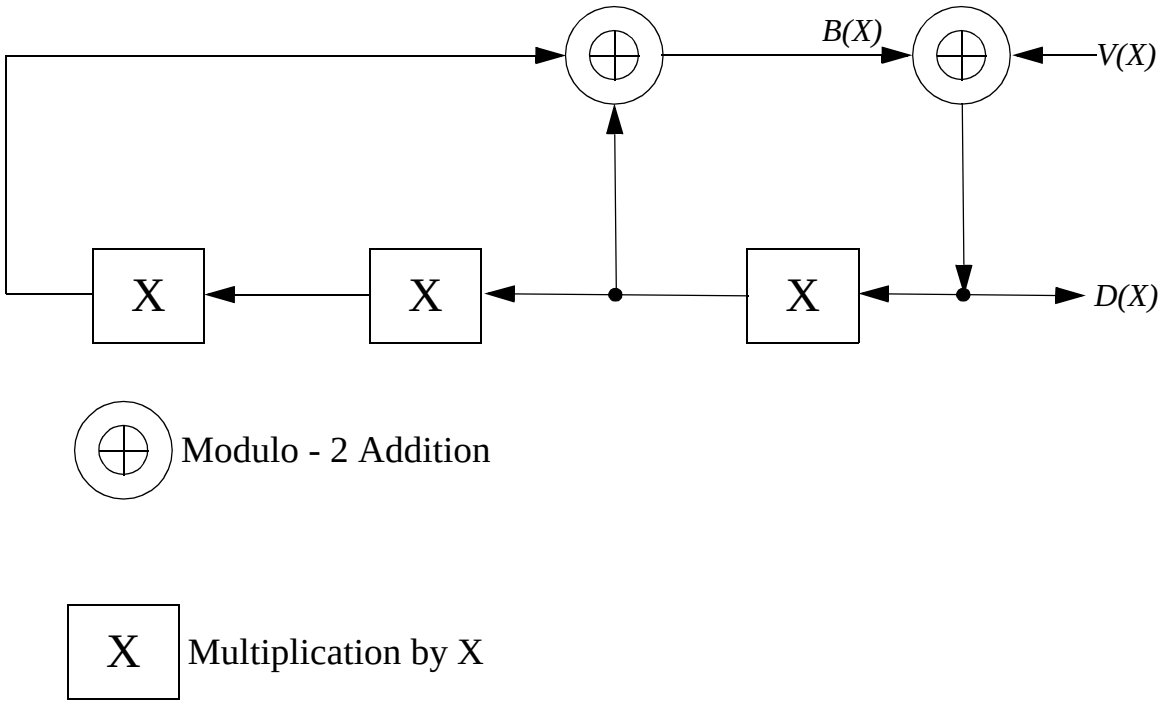



**Figure 1.13** An Example Division Circuit for Decoding Cyclic Codes [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 109].

where *Q(X)* is the quotient computed in the division process.

Multiplying both sides of the previous equation by *G(X)* yields

$$X^{n-k}D(X) = G(X)Q(X) + R(X)$$

or

$$X^{n-k}D(X) - R(X) = G(X)Q(X)$$

Recall, however, that all addition and subtraction operations are performed using modulo-2 arithmetic, so addition and subtraction are identical. Consequently,

$$X^{n-k}D(X) - R(X) = X^{n-k}D(X) + R(X) = G(X)Q(X) = V(X)$$

Therefore, the code polynomial, *V(X)*, formed as

$$V(X) = X^{n-k}D(X) + R(X)$$

is an exact multiple of the generator polynomial, *G(X)*.

**Arithmetic Codes.** *Arithmetic codes* are very useful when it is desired to check arithmetic operations such as addition, multiplication, and division [AVIZIENIS 71a]. The basic concept is the same as all coding techniques. The data presented to the arithmetic operation is encoded before the operations are performed. After completing the arithmetic operations, the resulting code words are checked to make sure that they are valid code words. If the resulting code words are not valid, an error condition is signaled.

An arithmetic code must be invariant to a set of arithmetic operations. An arithmetic code, *A*, has the property that *A(b*c)=A(b)*A(c)*, where *b* and *c* are operands, * is some arithmetic operation, and *A(b)* and *A(c)* are the arithmetic code words for the operands *b* and *c*, respectively. Stated verbally, the performance of the arithmetic operation on two arithmetic code words will produce the arithmetic code word of the result of the arithmetic operation. To completely define an arithmetic code, the method of encoding and the arithmetic operations for which the code is invariant must be specified. The most common examples of arithmetic codes are the *AN* codes, residue codes, and the inverse residue codes.

The simplest arithmetic code is the *AN code* which is formed by multiplying each data word, *N*, by some constant, *A*. The *AN* codes are invariant to addition and subtraction but not multiplication and division. If $N_1$ and $N_2$ are two operands to be encoded, the resulting code words will be $AN_1$ and $AN_2$, respectively. If the two code words are added, the sum is $A(N_1 + N_2)$, which is the code word of the correct sum. The operations performed under an *AN* code can be checked by determining if the results are evenly divisible by the constant, *A*. If the results are not evenly divisible by *A*, an error condition is signaled.

The magnitude of the constant, *A*, determines both the number of extra bits required to represent the code words and the error detection capability provided. The selection of the constant, *A*, is critical to the effectiveness and the efficiency of the resulting code. First, for binary codes, the constant must not be a power of two. To see the reason for this limitation, suppose that we encode the binary number, ($a_{n-1}\ a_{n-2}$ ... $a_2\ a_1\ a_0$ ) by multiplying by the constant, $A=2^a$. Multiplication by $2^a$ is equivalent to a left arithmetic shift of the original binary word, so the resulting code word will be ($a_{n-1}\ a_{n-2}$ ... $a_2\ a_1\ a_0$ 0 0 ... 0 ), where *a* 0s have been appended to the original binary number. The decimal representation of the code word is given by

$$a_{n-1}2^{a+n-1} + \ldots + a_2 2^{a+2} + a_1 2^{a+1} + a_0 2^a + 02^{a-1} + \ldots + 02^1 + 02^0$$

which is clearly, evenly divisible by $2^a$. It is also easy to see, however, that changing just one coefficient will still yield a result that is evenly divisible by $2^a$. For example, if the coefficient of the $2^a$ term changes from 0 to 1, the result will remain evenly divisible by $2^a$. Thus, an *AN* code that has $A=2^a$ is not capable of detecting single-bit errors.

An example of a valid *AN* code is the 3*N* code where all words are encoded by multiplying by a factor of three. If the original data words are *n* bits in length, the code words for the 3*N* code will require *n*+2 bits. The encoding of operands in the 3*N* code can be performed by a simple addition if we recognize that we can multiply any number by three by adding the original number to a value that is twice that number. In other words, we form 3*N* by adding *N* and 2*N*. The quantity, 2*N*, is easily created by shifting the binary number left by one place. The numbers, *N* and 2*N*, can then be added.

The next example class of arithmetic codes is the residue codes. A residue code is a separable

arithmetic code created by appending the residue of a number to that number. In other words, the code word is constructed as $D|R$, where $D$ is the original data and $R$ is the residue of that data. The encoding operation consists of determining the residue and appending it to the original data. The decoding process involves simply removing the residue, thus leaving the original data word.

The residue of a number is simply the remainder generated when the number is divided by an integer. For example, suppose we have an integer, $N$, and we divide $N$ by another integer, $m$. $N$ may be written as an integer multiple of $m$ as

$$N = Im + r$$

or

$$\frac{N}{m} = I + \frac{r}{m}$$

where $r$ is the remainder, sometimes called the residue, and $I$ is the quotient. The quantity $m$ is called the check base, or the modulus. For example, if $N$=14 and $m$=3, the quotient, $I$, will be four and the residue will be two. We often write this as

$$14 = 2 modulo (3)$$

Separable residue codes, as mentioned previously, are formed by appending the residue of a data word to that data word. The number of extra bits required to represent the code word depends on the particular modulus selected. The residue will never be larger than the modulus; in fact

$$0 \leq r < m$$

For example, if the original data is $n$ bits and the modulus is three, the code word will require $n$+2 bits.

The primary advantages of the residue codes are that they are invariant to the operation of addition, and the residues can be handled separately from the data during the addition process. The basic structure of an adder that uses the separable residue code for error detection is shown in Figure 1.14. The two data words, $D_1$ and $D_2$, are added to form a sum word, $S$. The residues, $r_1$ and $r_2$, of $D_1$ and $D_2$, respectively, are also added using a modulo-$m$ adder, where m is the modulus used

to encode $D_1$ and $D_2$. If the operations are performed correctly, the modulo-$m$ addition of $r_1$ and $r_2$ will yield the residue, $r_s$, of the sum, $S$. A separate circuit is then used to calculate the residue of $S$. If the calculated residue, $r_c$, differs from $r_s$, an error has occurred in one part of the process. For example, errors can be detected that occur in the generation of $S$, $r_s$, or $r_c$.

If the modulus for the residue code is selected in a special manner, a so-called *low-cost residue code* results. Specifically, *low-cost residue codes* have a modulus of $m=2^b-1$, where $b$ is some integer greater than or equal to two. The number of extra bits required in a low-cost residue code is equal to $b$. The main advantage of the low-cost residue code is the ease with which the encoding process can be performed. Recall that we must determine a remainder to encode information using a residue code; therefore, a division is necessary. The low-cost residue codes, however, allow the division to be recast as an addition process. The information bits to be encoded, are first divided into groups, each group containing $b$ bits. The groups are then added in a modulo-$(2^b-1)$ fashion to form the residue of the information bits.

A modification of the separable residue code is the separable *inverse-residue code*. The *inverse-residue code* is formed in a manner similar to that of the residue code by appending information to the original data. Rather than append the residue, the inverse residue is calculated and appended. The inverse residue, $Q$, is calculated for a data word, $N$, as $m$-$r$, where $m$ is the modulus and $r$ is the residue of $N$. The code word for $N$ then becomes $N|Q$.

The inverse-residue codes have been found to have better fault detection capability for so-called repeated-use faults [AVIZIENIS 71a]. A *repeated-use fault* is one that is encountered multiple times before the code is checked because the hardware is used multiple times before the code is checked. For example, if repeated addition is used to perform multiplication and the adder has a fault of some type, a repeated-use fault will occur. Repeated-use faults are particularly difficult to detect because subsequent effects of the fault can cancel the previous effects of the fault, thus rendering the fault undetectable.

**Berger Codes**. A very simple form of coding is the *Berger code* [LALA 85]. Berger codes are formed by appending a special set of bits, called the check bits, to each word of information. Therefore, the Berger code is a separable code. The check bits are created based on the number of

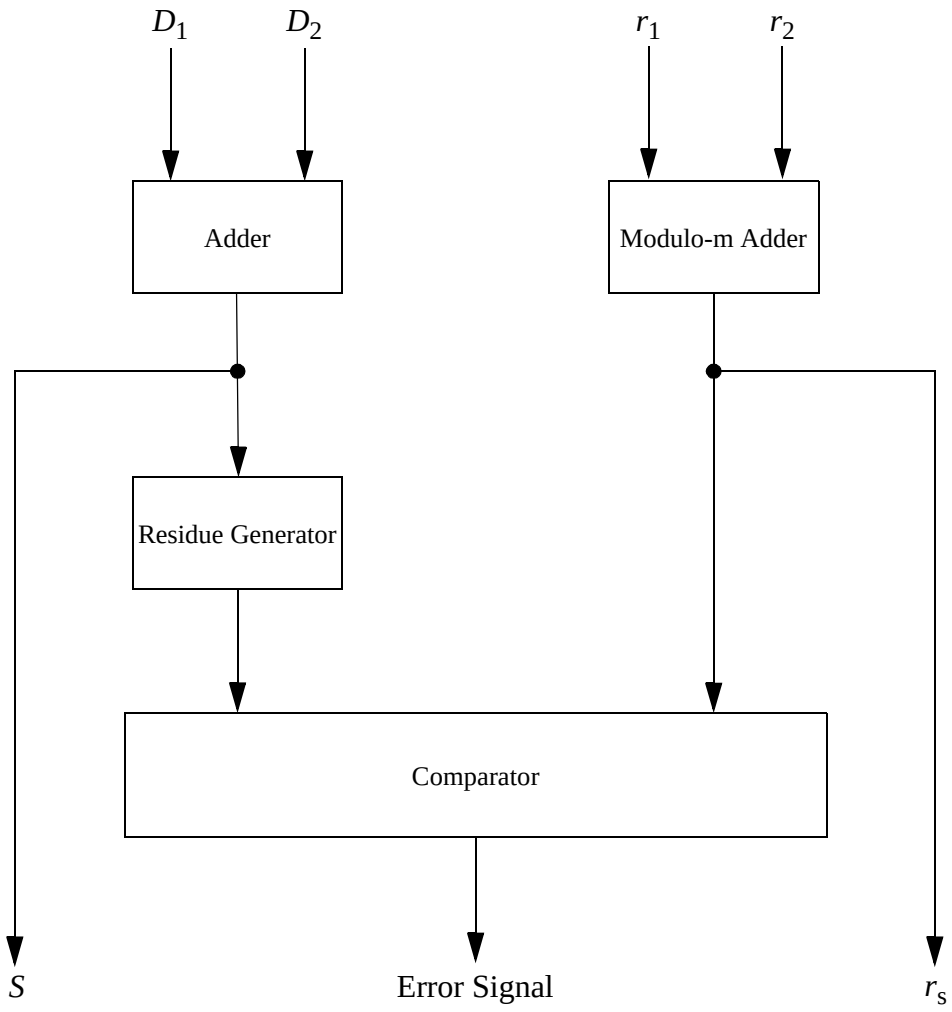



**Figure 1.14** An Adder Using a Residue Code for Error Detection [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 118].

1s in the original information. A Berger code of length $n$ will have $I$ information bits and $k$ check bits where

$$k = \lceil \log(I + 1) \rceil$$

and

$$n = I + k$$

A code word is formed by first creating a binary number that corresponds to the number of 1s in

the original I bits of information. The resulting binary number is then complemented and appended to the *I* information bits to form the (*I*+*k*)-bit code word. For example, suppose that the information to be encoded is (0111010), such that *I*=7. The value of *k* is then

$$k = \lceil \log(7 + 1) \rceil = 3$$

The number of 1s in this word of information is four, and the three-bit binary representation of four is (100). The complement of (100) is (011), so the resulting code word is (0111010011) which is simply the original information with 011 appended.

If the number of information bits is related to the number of check bits by the relationship

$$I = 2^k - 1$$

the resulting code is called a *maximal length Berger code*. For example, the code constructed for *I*=7 and *k*=3 is a maximal length Berger code. The primary advantages of the Berger codes are that they are separable and they detect all multiple, unidirectional errors. For the error detection capability provided, the Berger codes use the fewest number of check bits of the available separable codes [LALA 85].

**Hamming Error Correcting Codes.** Possibly the most common extension of the fundamental parity approach is the Hamming error-correcting code [HAMMING 50]. Many memory designs incorporate error correction for several reasons. First, Hamming error correction is relatively inexpensive; typically, the Hamming codes require anywhere from ten to forty percent redundancy. Second, the Hamming codes are efficient in terms of the time required to perform the correction process; the encoding and the decoding processes inject relatively small time delays. Third, the error correction circuit is readily available on inexpensive chips. Finally, the memory can contribute as much as 60 to 70% of the faults in a system. In addition, transient faults are becoming much more prevalent as memory chips become denser. The combination of permanent and transient faults in memories makes the use of error correction very attractive.

The Hamming codes are best thought of as overlapping parity. As we saw in the overlapping parity approach, the check bits provide one unique combination for each possible information bit that can be erroneous, one combination for each parity check bit that can be erroneous, and one

combination for the error-free case. The Hamming code is formed by partitioning the information bits into parity groups and specifying a parity bit for each group. The ability to locate which bit is faulty is obtained by overlapping the groups of bits. In other words, a given information bit will appear in more than one group in such a way that if the bit is erroneous, the parity bits that are in error will identify the erroneous bit.

The basic process involved in the Hamming codes is no different from that of other codes. First, the original data is encoded by generating a set, call it $C_g$, of parity check bits. When it is desired to check the information for its correctness, the encoding process is repeated and a set, call it $C_c$, of parity check bits is regenerated. If $C_g$ and $C_c$ agree, the information is assumed to be correct. If, however, $C_g$ and $C_c$ disagree, the information is incorrect and must be corrected. To aid in the correction, we define the syndrome, $S$, as the result obtained by forming the EXCLUSIVE-OR of $C_g$ and $C_c$. The syndrome is a binary word that is 1 in each bit position in which $C_g$ and $C_c$ disagree. A syndrome that is all 0s is indicative of correct information.

The basic structure of a memory that uses the Hamming single error correcting code is shown in Figure 1.15. When data is written to memory, the check bits are generated and stored in memory along with the original information. Upon reading the information and the check bits from memory, the check bits are regenerated and compared to the stored check bits to generate the syndrome. The syndrome is then decoded to determine if a bit is erroneous, and if so, the erroneous bit is corrected by complementation. The corrected data is then passed to the user of the memory. Memories that use this type of correction are usually designed such that data is corrected without interrupting the normal operation of the system. The user of the memory might be informed, however, that a correction has occurred such that maintenance can be performed if corrections are continually required.

The basic Hamming code described above provides for the correction of single-bit errors. Unfortunately, double-bit errors will be erroneously corrected using the basic Hamming code. To overcome the problem of erroneous correction and provide a code that can correct single-bit errors and identify double-bit errors, the basic Hamming code is modified. The resulting code is called the modified Hamming code. The modification consists of simply adding one additional parity check bit that checks parity over the entire Hamming code word. If a single bit is in error, the addi-

tional parity bit will indicate that the overall parity is incorrect. The syndrome will then point to the bit that is erroneous. If a double-bit error occurs, the additional parity bit will indicate that the overall parity is correct because a single parity check cannot detect a double-bit error. But, the syndrome will be nonzero because the remaining parity checks will indicate an error. Therefore, the double error can be detected, and an erroneous correction prevented. To summarize, if the overall parity is incorrect and the syndrome is not 0, a single-bit error is corrected. If the overall parity is correct and the syndrome is not 0, a double-bit error is identified and no correction occurs. If the overall parity is correct and the syndrome is 0, the data is assumed to be correct.

**Self-checking Concepts.** The concept of *self-checking logic* has increased in popularity because of the traditional problem of "checking the checker". In many designs that use coding schemes or duplication with comparison it is necessary to compare the outputs of two modules or to verify that the output is a valid codeword. The basic problem with such techniques, as we have seen, is the reliance of the approaches on the correct operation of comparators or code checkers. If the code checker fails, for example, the system can indicate that an error exists when in fact one does not, or the system can fail to detect a legitimate error that occurs. In many applications either condition is unacceptable. One possible solution is to design comparators and code checkers that are capable of detecting their own faults. Consequently, the concept of *self-checking logic* has been developed. Before beginning the discussions of self-checking logic, we must first introduce several importants terms that are crucial to the understanding of self-checking technology.

In general, a circuit is said to be *self-checking* if it has the ability to automatically detect the existence of a fault without the need for any externally applied stimulus [LALA 85]. In other words, a self-checking circuit determines if it contains a fault during the normal course of its operations. Self-checking logic is typically designed using coding techniques similar to those discussed already. The basic idea is to design a circuit that when fault-free and presented a valid input code word will produce the correct output code word. If a fault exists, however, the circuit should produce an invalid output code word so that the existence of the fault can be detected.

To formalize the concept of self-checking logic, we will define the terms *fault secure*, *self-testing*, and *totally self-checking*. In presenting each definition it is important to understand that we

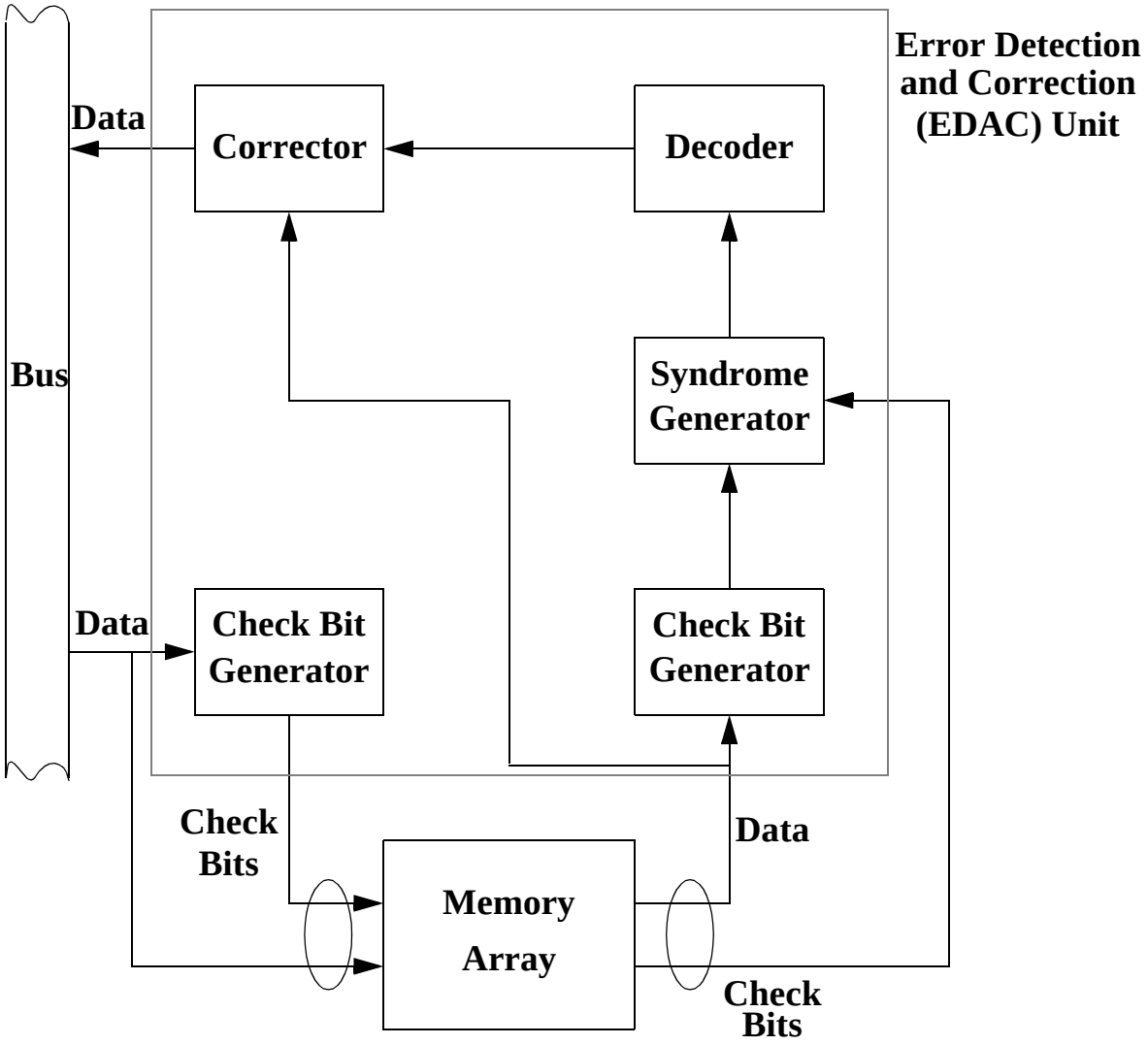



**Figure 1.15** Memory Organization Using an Error-correcting Code [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 132].

are considering circuits designed to accept code words on their input lines and produce code words on their output lines.

A circuit is said to be *fault secure* if any single fault within the circuit results in that circuit either producing the correct code word or producing a noncode word, for any valid input code word [LALA 85]. In other words, a circuit is fault secure if the fault either has no affect on the output or

the output is affected such that it becomes an invalid code word. A circuit would not be fault secure, for example, if a fault resulted in the output becoming incorrect but still a valid code word.

A circuit is said to be *self-testing* if there exists at least one valid input code word that will produce an invalid output code word when a single fault is present in the circuit [LALA 85]. In other words, a circuit is self-testing if each single fault is detectable since a fault that resulted in valid output code words for each possible input code word would be undetectable.

Finally, a circuit is said to be *totally self-checking* if it is both fault secure and self-testing [LALA 85]. The fault secure property guarantees that the circuit will either produce the correct code word output or an invalid code word output when any single fault occurs. The self-testing property guarantees that there is at least one input code word that will produce an invalid code word output from the circuit when a fault is present. In summary, a circuit is totally self-checking if all single faults are detectable by at least one valid code word input, and when a given input combination does not detect the fault, the output is the correct code word output.

Another way of looking at these basic concepts is as follows. The primary inputs of the circuit are encoded to produce the set of valid input code words that is a subset of the total set of inputs. Similarly, the total set of output values is partitioned into valid output code words and is further partitioned into the correct output code words. During normal operation, a fault-free circuit will accept a valid input code word and produce the correct output code word. A fault secure circuit will accept a valid input code word, and, when a fault is present, produce either the correct output code word or a noncode word. A self-testing circuit will produce correct code word outputs, valid but incorrect code word outputs, or completely invalid code word outputs. However, for any single fault that can be present, you are guaranteed that there is at least one valid input code word that will result in the output being an invalid code word. Finally, a totally self-checking circuit will always produce either the correct code word at the output or an invalid code word. Also, there will be at least one valid input code word that will result in an invalid output code word when any single fault is present.

The general structure of a totally self-checking (TSC) circuit is shown in Figure 1.16. During normal operation, coded inputs are applied to the functional circuit and coded outputs are produced

at the circuit's output. A checker verifies that the outputs are indeed valid code words and provides an error indication if they are not. To provide a truly TSC design, both the functional circuit and the checker must possess the TSC property. Perhaps the key to the correct operation of the circuit is the TSC checker, so we will consider its operation.

The function of the checker is to determine if the output of the functional circuit is a valid code word or not. In addition, the checker must indicate if any faults have occurred in the checker itself. To accomplish both tasks, the output of the checker is encoded to produce a coded error signal. Rather than have a single-bit output that provides a "faulty" or "not faulty" indication, the output consists of two bits that are: (1) complementary if the input to the checker is a valid code word *and* the checker is fault-free, or (2) noncomplementary if the input to the checker is not a valid code word *or* the checker contains a fault. One obvious reason for using two checker outputs is to overcome the problem of the checker output becoming *stuck* at either the logic 0 or the logic 1 value.

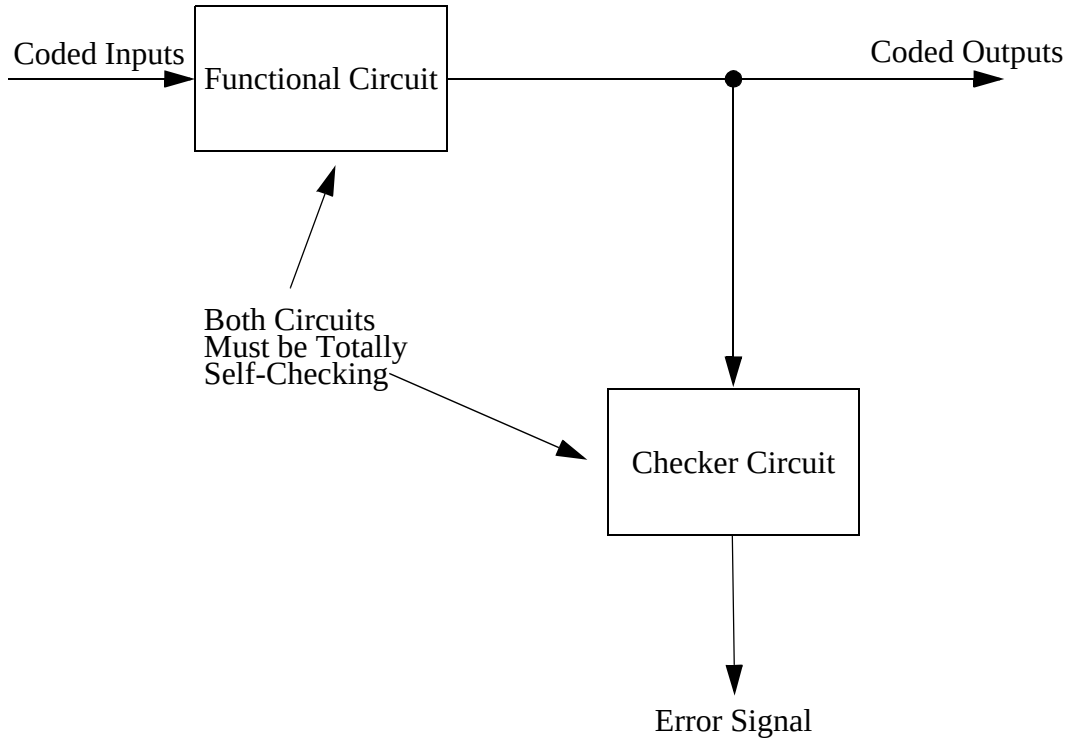



**Figure 1.16** Basic Structure of a Totally Self-checking Circuit [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 397].

The checker must possess the *code disjoint* property [LALA 85]. Code disjoint implies that when the checker is fault-free, valid code words on the checker's input lines must be mapped into valid error codes (the checker's outputs are complementary) on the checker's output lines. Likewise, invalid code words on the checker's input lines must be mapped into invalid error code words (the checker's outputs are not complementary) on the checker's outputs.

### 1.2.3. Time Redundancy

The fundamental problem with the forms of redundancy discussed thus far is the penalty paid in extra hardware for the implementation of the various techniques. Both hardware redundancy and information redundancy can require large amounts of extra hardware for their implementation. In an effort to decrease the hardware required to achieve fault detection or fault tolerance, time redundancy has recently received much attention. Time redundancy methods attempt to reduce the amount of extra hardware at the expense of using additional time. In many applications, the time is of much less importance than the hardware because hardware is a physical entity that impacts weight, size, power consumption and cost. Time, on the other hand, may be readily available in some applications. It is important to understand that the selection of a particular type of redundancy is very dependent upon the application. For example, some systems can better stand additional hardware than additional time; others can tolerate additional time much easier than additional hardware. The selection in each case must be made by examining the requirements of the application and the available techniques that can meet such requirements.

**Transient Fault Detection.** The basic concept of time redundancy is the repetition of computations in ways that allow faults to be detected. Time redundancy can function in a system in several ways. The fundamental concept is to perform the same computation two or more times and compare the results to determine if a discrepancy exists. If an error is detected, the computations can be performed again to see if the disagreement remains or disappears. Such approaches are often good for detecting errors resulting from transient faults but cannot provide protection against errors resulting from permanent faults.

The main problem with many time redundancy techniques is assuring that the system has the same data to manipulate each time it redundantly performs a computation. If a transient fault has

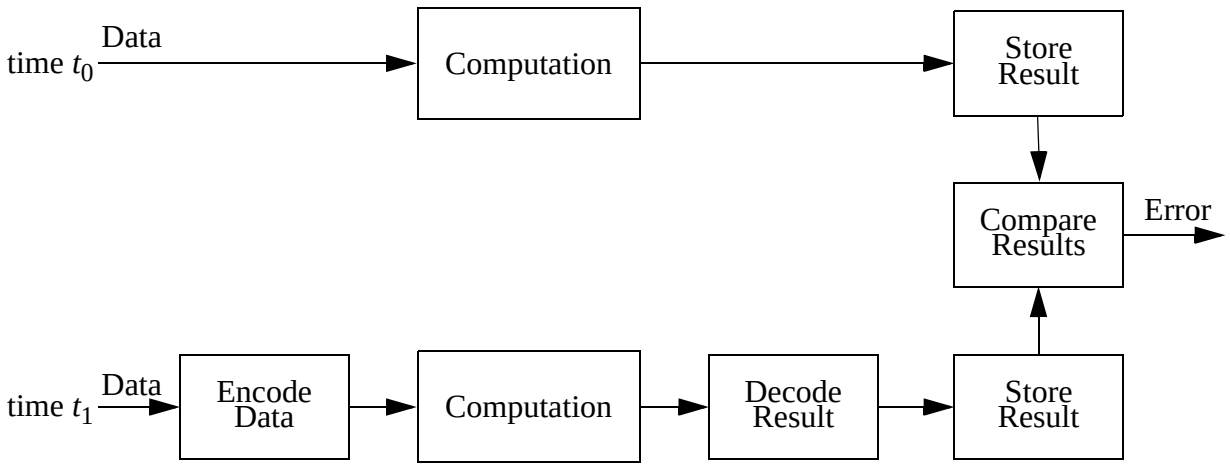



**Figure 1.17** Permanent Fault Detection Using Time Redundancy [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 137].

occurred, a system's data may be completely corrupted, making it difficult to repeat a given computation.

**Permanent Fault Detection.** In the past, time redundancy has been used primarily to detect transients in systems. One of the biggest potentials of time redundancy, however, now appears to be the ability to detect permanent faults while using a minimum of extra hardware. The fundamental concept is illustrated in Figure 1.17. During the first computation or transmission, the operands are used as presented and the results are stored in a register. Prior to the second computation or transmission, the operands are encoded in some fashion using an encoding function. After the operations have been performed on the encoded data, the results are then decoded and compared to those obtained during the first operation. The selection of the encoding function is made so as to allow faults in the hardware to be detected. Example encoding functions might include the complementation operator and an arithmetic shift.

The complementation operator has been applied to the transmission of digital data over wire media and the detection of faults in digital circuits. Suppose that we wish to detect errors in data that is transmitted over a parallel bus using the time redundancy approach. At time $t_0$ we transmit the original data, and at time $t_0 + \Delta$ we transmit the complement of the data. If a line of the bus is

stuck at either a 1 or a 0, the two versions of the information that are received will not be complements of each other. Therefore, the fault can be detected. In general, if a sequence of information is transmitted using this approach, each bit line should alternate between a logic 1 and a logic 0; provided the transmission is error free. Thus, the reason for the name *alternating logic* being applied to this approach.

The concept of alternating logic can be applied to general, combinational logic circuits if the circuit possesses the property of self duality. A combinational circuit is said to be *self dual* if and only if $f(X) = \overline{f(\overline{X})}$, where $f$ is the output of the circuit and $X$ is the input vector for the circuit. Stated verbally, a combinational circuit is self dual if the output for the input vector, $X$, is the complement of the output when the input vector, $\overline{X}$, is applied. For a self-dual circuit, the application of an input, $X$, followed by the input, $\overline{X}$, will produce outputs that alternate between 1 and 0. The key to the detection of faults using the alternating logic approach is determining that at least one input combination exists for which the fault will not result in alternating outputs.

A key advantage of the alternating logic approach is that any combinational circuit with $n$ input variables can be transformed into a self-dual circuit with no more than $n$+1 input variables. To see this we first define the dual of a function. The dual, $f_d$, of an $n$-variable function, $f$, is given by

$$f_d = \overline{f(\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n})}$$

In other words, the dual of the function $f$ is obtained by first complementing $f$ and then replacing each variable with the complement of the variable. The function $f_{sd}$ given by

$$f_{sd} = x_{n+1} f + \overline{x_{n+1}} f_d$$

is then a self-dual function.

The use of alternating logic will detect a set of faults if for every fault within the set there is at least one input combination that produces non-alternating outputs. It is important to note, however, that faults may not be immediately detected using alternating logic. Depending upon the application, the time elapsed before the detection of the fault may or may not be significant.

Another form of encoding function is called recomputing with shifted operands (RESO)

[PATEL 82]. RESO was developed as a method to provide concurrent error detection in arithmetic logic units (ALUs). RESO uses the basic time redundancy method that was shown in Figure 1.17, and the encoding function is selected as the left shift operation with the decoding function being the right shift operation. In many cases, the shift operations can be either arithmetic or logical shifts. The RESO technique was derived assuming bit-sliced organizations of the hardware.

In logical operations, it is relatively easy to understand the error detection capability of the RESO technique. Suppose that bit slice *i* of a circuit is faulty and produces an erroneous value for the function's output at that bit slice. During the first computation with the operands not shifted, the $i^{th}$ output of the circuit will be erroneous. When the operands are shifted by one bit, the faulty bit slice will then operate on, and corrupt, the $(i\text{-}1)^{th}$ bit. When the result is shifted back to the right, the two results -- the first with unshifted operands and the second with shifted operands -- will either both be correct or they will disagree in either (or both) the $i^{th}$ or the $(i\text{-}1)^{th}$ bits. If only one bit slice is faulty and that faulty slice has no impact on any other bit slices, a single left shift will detect the errors that occur in logical operations.

For a bit-sliced, ripple-carry adder, a two-bit arithmetic shift is required to guarantee the detection of errors that can occur [PATEL 82]. Once again suppose that bit slice *i* is faulty. In a ripple-carry adder, a faulty bit slice can have one of three effects; the carry out of the bit slice can be erroneous, the sum bit out of the bit slice can be erroneous, or both may be in error. If the sum bit is 0 when it should be 1, the resulting sum will be in error by $-2^i$. In other words, the sum will be $2^i$ smaller than it should be. If the sum bit is 1 when it should be 0, the sum will be $2^i$ larger than it should be. If the carry bit out of the bit slice is affected, the sum bit in position $(i+1)$ will be impacted. If the carry bit is 0 when it should be 1, the sum will be decreased by $2^{i+1}$. If the carry bit is 1 when it should be 0, the sum will be increased by $2^{i+1}$. If both the sum bit and the carry bit are affected, the resulting sum can be in error in one of four ways; the sum and carry are both erroneously 1, the sum and carry are both erroneously 0, the sum is erroneously 1 and the carry is erroneously 0, and the sum is erroneously 0 and the carry is erroneously 1. When both the sum and the carry are erroneously 1, the resulting sum will be increased by $2^{i+1}+2^i = (3)2^i$. If both the sum and the carry are erroneously 0, the resulting sum will be decreased by $(3)2^i$. If the carry is erroneously 1 and the sum erroneously 0, the resulting sum will be increased by $2^{i+1}$ because of the carry and

decreased by $2^i$ because of the sum. The overall impact is a decrease of $2^i$, as was the case when only the sum bit was erroneously 0. Finally, if the carry is erroneously 0 and the sum is erroneously 1, the resulting sum will be decreased by $2^{i+1}$ because of the carry-bit error and increased by $2^i$ because of the sum-bit error. The net effect is a decrease of $2^i$. In summary, the result generated for the unshifted operands if bit *i* is faulty will be incorrect by one of [0, $+-2^i$ , $+-2^{i+1}$ , $+-(3)2^i$].

When the operands are shifted to the left by two bits, a similar analysis can show that the result will be incorrect by one of [0, $+-2^{i-2}$ , $+-2^{i-1}$ , $+-(3)2^{i-2}$]. As can be seen, the results of the two computations cannot agree unless both are correct. Therefore, the error will be detected.

Time redundancy techniques form an important class of options for designing fault-tolerant systems. Just as all redundancy approaches, however, time redundancy cannot be used in all applications because of the additional time that must be employed. If time is available, however, time redundancy techniques do provide an opportunity to minimize the amount of additional hardware required.

**Recomputation for Error Correction.** The time redundancy approach can also provide for error correction if the computations are repeated three or more times. Consider, for example, a logical AND operation. Suppose the operation is performed three times; first, without shifting the operands; second, with a one-bit, logical shift of the operands; and third, with a two-bit, logical shift of the operands. The results generated using the shifted operands are then shifted the appropriate number of bits to the right to properly position the bits of the results. Because each of the three operations used operands that were displaced from each other by at least one bit position, a different bit in each result will be affected by the faulty bit slice. If the bits in each position are then compared, the results due to the faulty bit slice can be corrected by performing a majority vote on the three results obtained for each bit position. Unfortunately, the approach described above will not work for arithmetic operations because the adjacent bits are not independent. A single, faulty bit slice can affect more than one bit of the result.

### 1.2.4. Software Redundancy

In applications that use computers, many fault detection and fault tolerance techniques can be implemented in software. The redundant hardware necessary to implement the capabilities can be

minimal, while the redundant software can be substantial. Redundant software can occur in many forms; you do not have to replicate complete programs to have redundant software. Software redundancy can appear as several extra lines of code used to check the magnitude of a signal or as a small routine used to periodically test a memory by writing and reading specific locations. In this section, we will consider several major software redundancy techniques; consistency checks, capability checks, and software replication methods [CHEN 78].

**Consistency Checks.** A consistency check uses *a priori* knowledge about the characteristics of information to verify the correctness of that information. For example, in some applications, it is known in advance that a digital quantity should never exceed a certain magnitude. If the signal exceeds that magnitude then an error of some sort is present. A consistency check can often be implemented easily in hardware but is most likely to appear in the software of a system. For example, a processing system can sample and store many sensor readings in a typical control application. Each sensor reading can be checked to verify that it lies within an acceptable range of values. As another example, the amount of cash requested by a patron at a bank's teller machine should never exceed the maximum withdrawal allowed. Likewise, the address generated by a computer should never lie outside the address range of the available memory.

An example of consistency checking that can be performed in hardware is the detection of invalid instruction codes in computers. Many computers use *n*-bit quantities to represent $2^k$ possible instruction codes where $2^k < 2^n$. In other words, there are $2^n - 2^k$ instruction codes that are illegal. Each instruction code can be checked to verify that it is not one of the illegal codes. If an illegal code occurs, the processor can be halted to prevent an erroneous operation from occurring. This technique is particularly useful in detecting a *run-away* processor that is erroneously interpreting data as instructions.

Another form of consistency checking that can prove valuable in many control applications is to compare the measured performance of the system with some predicted performance. This technique is particularly useful in control applications where some dynamic system is under control. The dynamic system can be modeled and the predicted performance obtained from a software implementation of the model. The actual performance of the system can then be measured and compared with the model-predicted performance. Any significant deviations of the measured per-

formance from the predicted performance can be indicative of a fault. The difficulty with this approach is twofold. First, the model must be accurate if good results are to be obtained. Second, it is difficult to establish the level of deviation that will be allowed before an error is signaled. In some applications, the nonlinearity of a system can result in a linearized model deviating substantially from the actual performance under certain input conditions.

**Capability Checks.** Capability checks are performed to verify that a system possesses the capability expected. For example, you would like to know whether or not you have your complete memory available or if all the processors in your multiprocessor system are working properly. As another example, you might want to know if the ALU in your processor is working properly.

Several forms of capability checks exist. The first is a simple memory test. A processor can simply write specific patterns to certain memory locations and read those locations to verify that the data was stored and retrieved properly. In many cases, it is not necessary to write and read a large number of locations to achieve reasonably good fault coverage. The memory test can be a supplement to parity as protection against faults in the memory.

Another example of a capability check is a set of ALU tests. Periodically, a processor can execute specific instructions on specific data and compare the results to known results stored in a read only memory (ROM). This form of capability check can verify both the ALU and the memory in which the known results are stored. The instructions that are executed can consist of adds, multiplies, logical operations, and data transfers.

Another form of capability check consists of verifying that all processors in a multiple processor system are capable of communicating with each other. This can consist of periodically passing specific information from one processor to another. For example, each processor may be required to set a specific bit in a shared memory to indicate their capability to communicate with that memory, and, as a result, communicate with other processors through that memory.

***N* Self-Checking Programming.** The software redundancy techniques that we have considered thus far have been those that use extra, or redundant, software to detect faults that can occur in hardware. We have not yet considered approaches for detecting, or possibly tolerating, faults that can occur in the software of a system. Software faults are unusual entities. Software does not

break as hardware does, but instead software faults are the result of incorrect software designs or coding mistakes. Therefore, any technique that detects faults in software must detect design flaws. A simple duplication and comparison procedure will not detect software faults if the duplicated software modules are identical, because the design mistakes will appear in both modules.

The concept of *N* self-checking programming is illustrated in Figure 1.18. Essentially, *N* unique versions of the program are written, and each version includes its set of acceptance tests. The acceptance tests are essentially checks performed on the results produced by the program and may be created using consistency checks and capability checks, for example. The selection logic, which may be a program itself, chooses the results from one of the programs that passes the acceptance tests. This approach is analogous to the hardware technique known as hot standby sparing. Since each program is running simultaneously, the reconfiguration process can be very fast. Provided that the software faults in each version of the program are independent of those in any of the

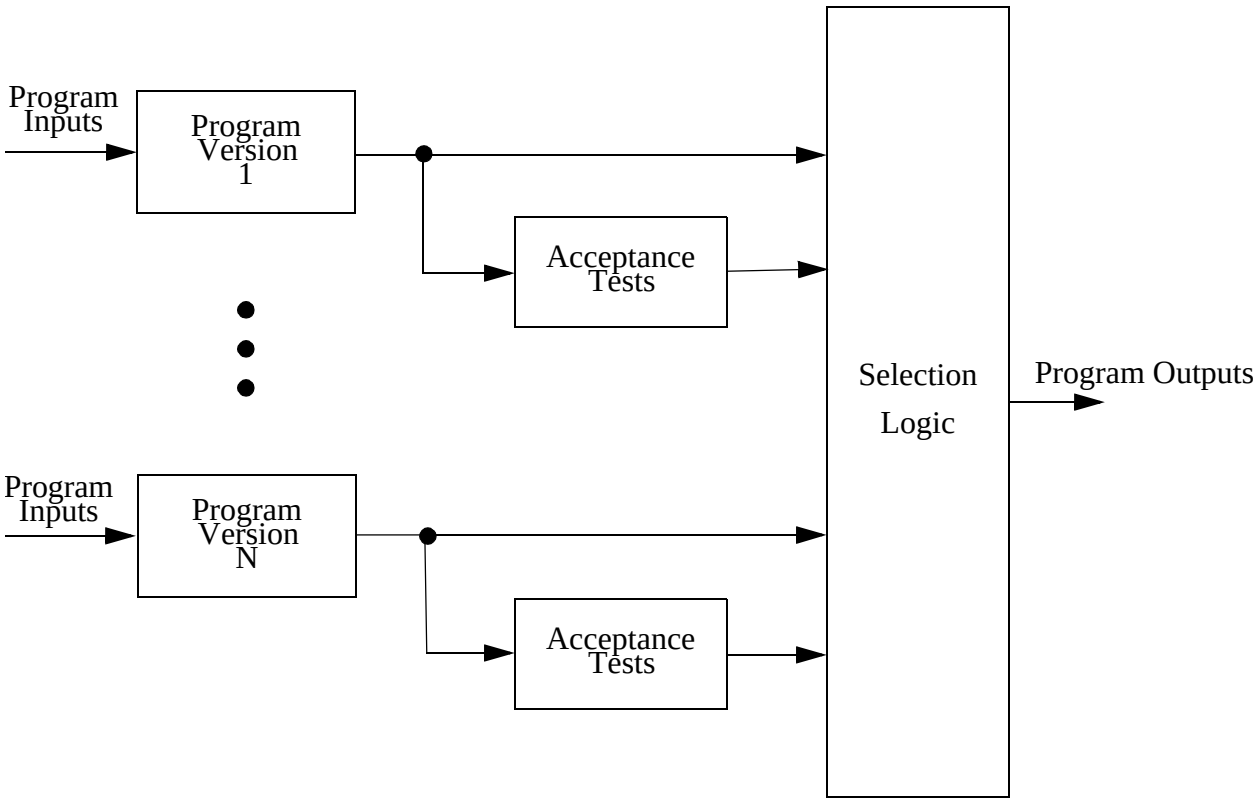



**Figure 1.18** The N self-checking programming approach to software fault tolerance.

other versions and the faults are detected as they occur by the acceptance tests then this approach can tolerate *N*-1 faults. It is important to note that the assumptions of fault independence and perfect fault coverage are very big assumptions to make in almost all applications.

***N*-version Programming.** The concept of *N*-version programming was developed to allow certain design flaws in software modules to be tolerated [CHEN 78]. The basic concept of *N*-version programming is to design and code the software module *N* times and to vote on the *N* results produced by these modules, as illustrated in Figure 1.19. Each of the *N* modules is designed and coded by a separate group of programmers. Each group designs the software from the same set of specifications such that each of the *N* modules performs the same function. However, it is hoped that by performing the *N* designs independently, the same mistakes will not be made by the different groups. Therefore, when a fault occurs, the fault will either not occur in all modules or it will occur differently in each module, so that the results generated by the modules will differ. Assuming that the faults are independent the approach can tolerate $\frac{N-1}{2}$ faults.

Certainly, the importance of software fault tolerance is easy to see. If we design a microprocessor-based system to be fault tolerant using a TMR configuration, the hardware redundancy will

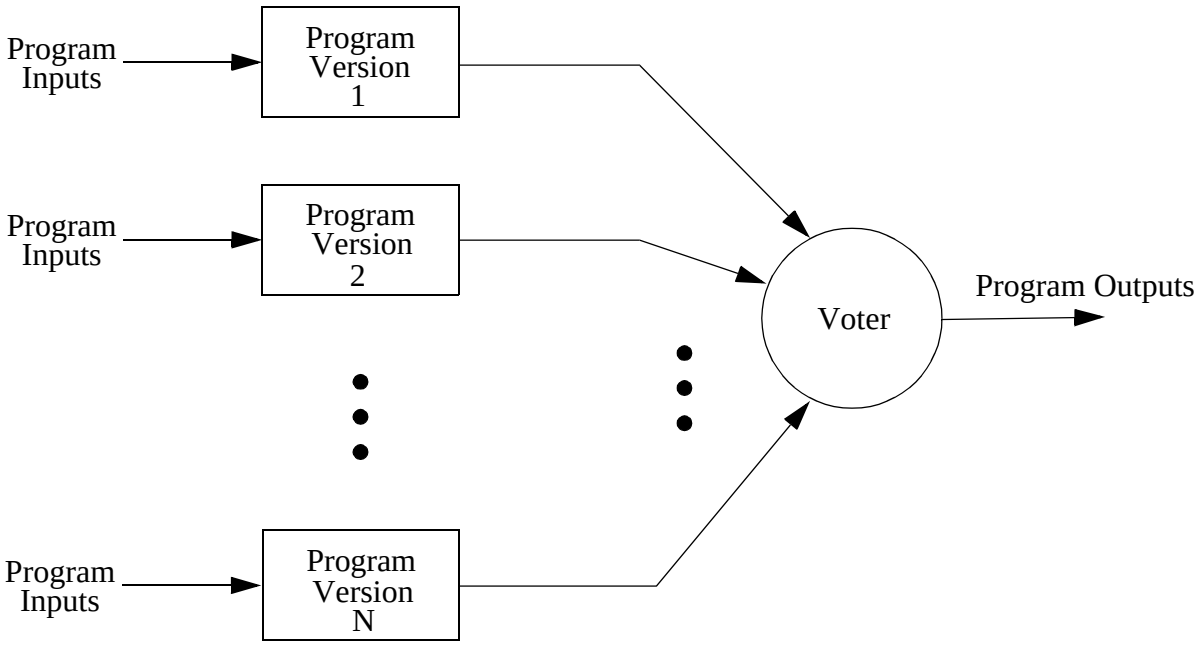



**Figure 1.19** The *N*-version programming concept.

be of little use if a single software fault can disable each of the redundant processors. The *N*-version programming technique states that each of the three processor's software should be designed and coded independently such that a common fault is less likely to occur.

The primary difficulties with the *N*-version approach are twofold. First, software designers and coders can tend to make similar mistakes. Therefore, we are not guaranteed that two completely independent versions of a program will not have identical faults. Second, the *N* versions of a program are still developed from a common specification, so the *N*-version approach will not allow the tolerance of specification mistakes.

To overcome many of the problems associated with *N*-version programming, software designers employ rigid design rules and methods to attempt to prevent faults from occurring. This approach we know as fault avoidance, and it is very important in the design of reliable software. If the software is designed correctly in the first place, fault tolerance techniques for the software will not be necessary.

**Recovery Blocks.** The recovery block approach to software fault tolerance is analogous to the active approaches to hardware fault tolerance, specifically the cold standby sparing approach. *N* versions of a program are provided, and a single set of acceptance tests is used. One version of the program is designated as the primary version, and the remaining *N*-1 versions are designated as spares, or secondary versions. The primary version of the software is always used unless it fails to pass the acceptance tests. If the acceptance tests are failed by the primary version then the first secondary version is tried. This process continues until one version passes the acceptance tests, or the system fails because none of the versions can pass the tests. The concept of the recovery block approach is illustrated in Figure 1.20. Assuming perfect coverage and independent faults, the approach can tolerant up to *N*-1 faults.

## 1.2.5. Redundancy Example

As an example of active redundancy consider the memory system which is illustrated in Figure 1.21 [CLARK 92]. The system was designed for satellite applications and uses two levels of active redundancy. The memory is organized into M modules, each of which is designed to store 256 megabits (Mb) of data, in this particular example. A total of $S_M$ spare memory modules is pro-

vided in the system, and each spare module can substitute for any of the M primary memory modules. Consequently, the memory system can tolerate up to $S_M$ complete memory module failures before the memory becomes inoperable. Each module also has internal redundancy, as will be discussed later, so the total number of memory integrated circuit (IC) faults that can be tolerated is greater than $S_M$ . It is important to note, however, that if we performed a reliability analysis of the system we would need to incorporate the concept of coverage at both levels of redundancy.

The organization of the memory control unit is shown in Figure 1.22. The memory control unit is responsible for performing the reconfiguration of the memory modules, if one of the modules is diagnosed as being failed. In other words, the memory control unit selects M of the $M+S_M$ modules for use in the memory. The memory control unit performs the top level of reconfiguration actions. The memory control unit is also responsible for the normal operations of the memory. Specifically, the memory control unit must partially decode addresses to determine which module should be enabled to respond to a particular memory address.

A more detailed organization of each module is shown in Figure 1.23. Each module uses column sparing as the means of providing spare columns of memory integrated circuits (ICs). If a

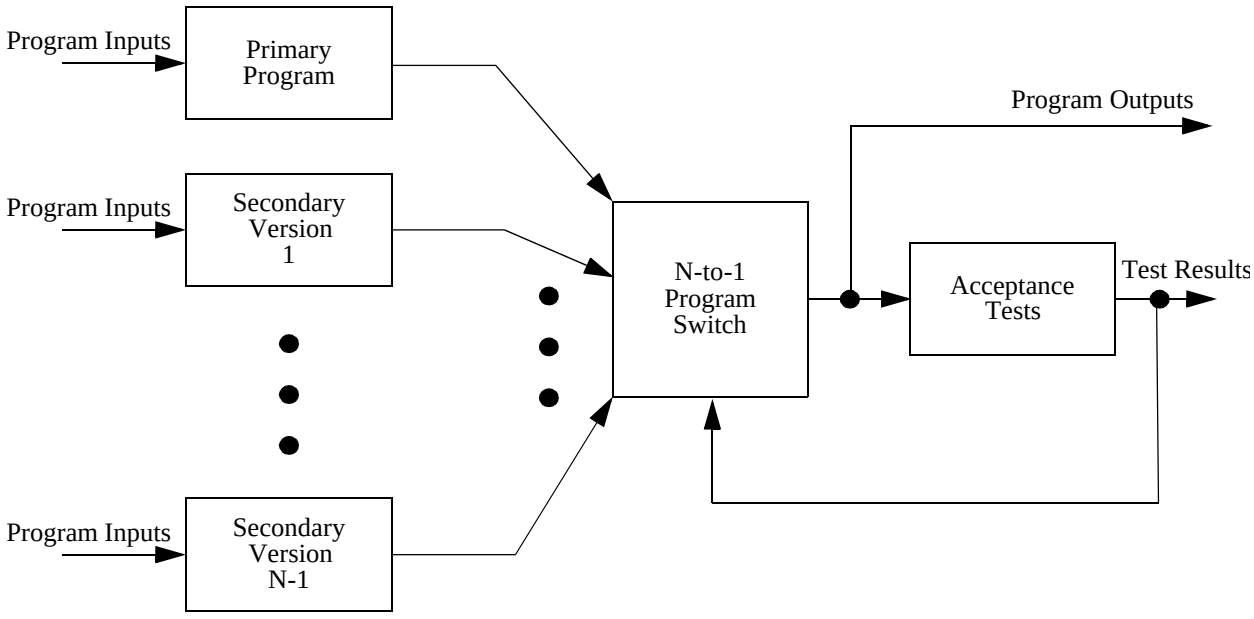



**Figure 1.20** The recovery block approach to software fault tolerance.

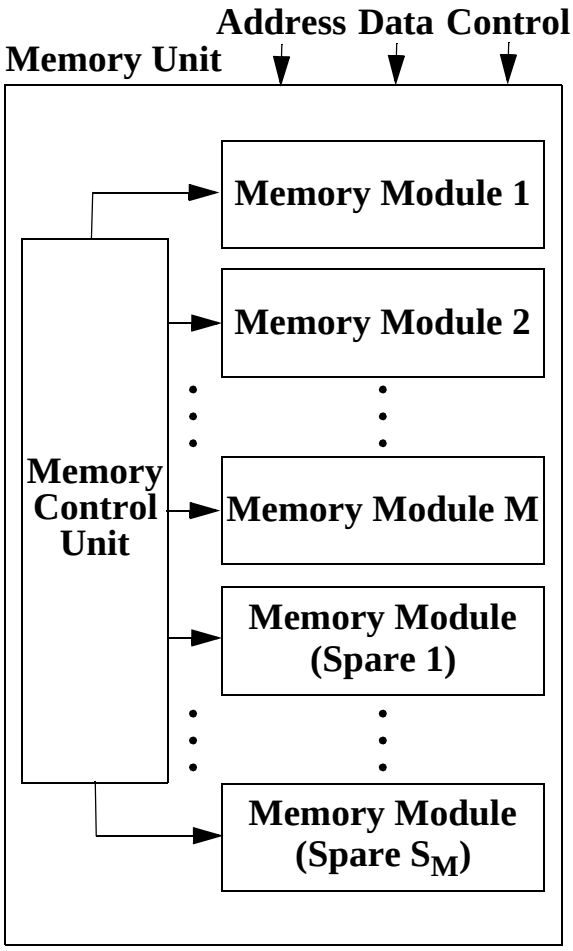



**Figure 1.21** Example of a memory using two-level active redundancy.

memory IC fails then the column containing that IC is eliminated from the system and replaced with a spare column. If the memory module runs out of spare columns then the entire module is replaced with a spare module. This type of redundancy is often referred to a two-level redundancy; the first level is the spare columns, and the second level is the spare memory modules. Both forms of reconfiguration are active techniques, and they require that the fault be detected, located, and successfully removed from the system. One of the primary fault detection mechanisms in this memory is the use of an error detecting and correcting code, as illustrated in Figure 1.23.

Figure 1.24 shows the organization of each column in the memory. The column control circuitry can be used to eliminate any specific column from the memory and replace that column with

one of the spare column. In other words, the column control circuitry directs the data to the active columns and prevents data from being directed to an inactive column.

## 1.3. Dependability Evaluation Techniques

In this section, we will introduce several approaches to quantitative evaluation, including the failure rate, mean time between failure (MTBF), mean time to failure (MTTF), fault coverage, reliability analysis, availability analysis, maintainability analysis, and safety analysis. Several techniques for generating the reliability, safety, and availability of a system will be presented.

### 1.3.1. Basic Definitions

This section will examine several basic definitions which are fundamental to quantitative evaluation techniques. Once the basic definitions are in place the various modeling approaches will be considered in more detail.

**Failure Rate and the Reliability Function.** Intuitively, the *failure rate* is the expected number of failures of a type of device or system per a given time period [SHOOMAN 68]. The failure

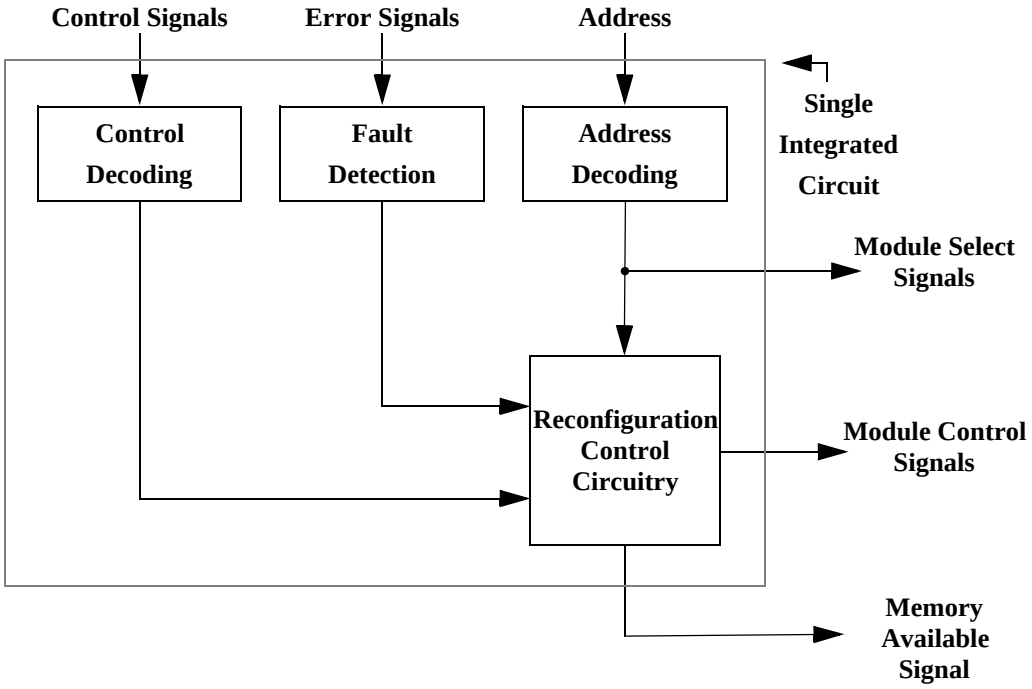



**Figure 1.22** Organization of the memory control unit.

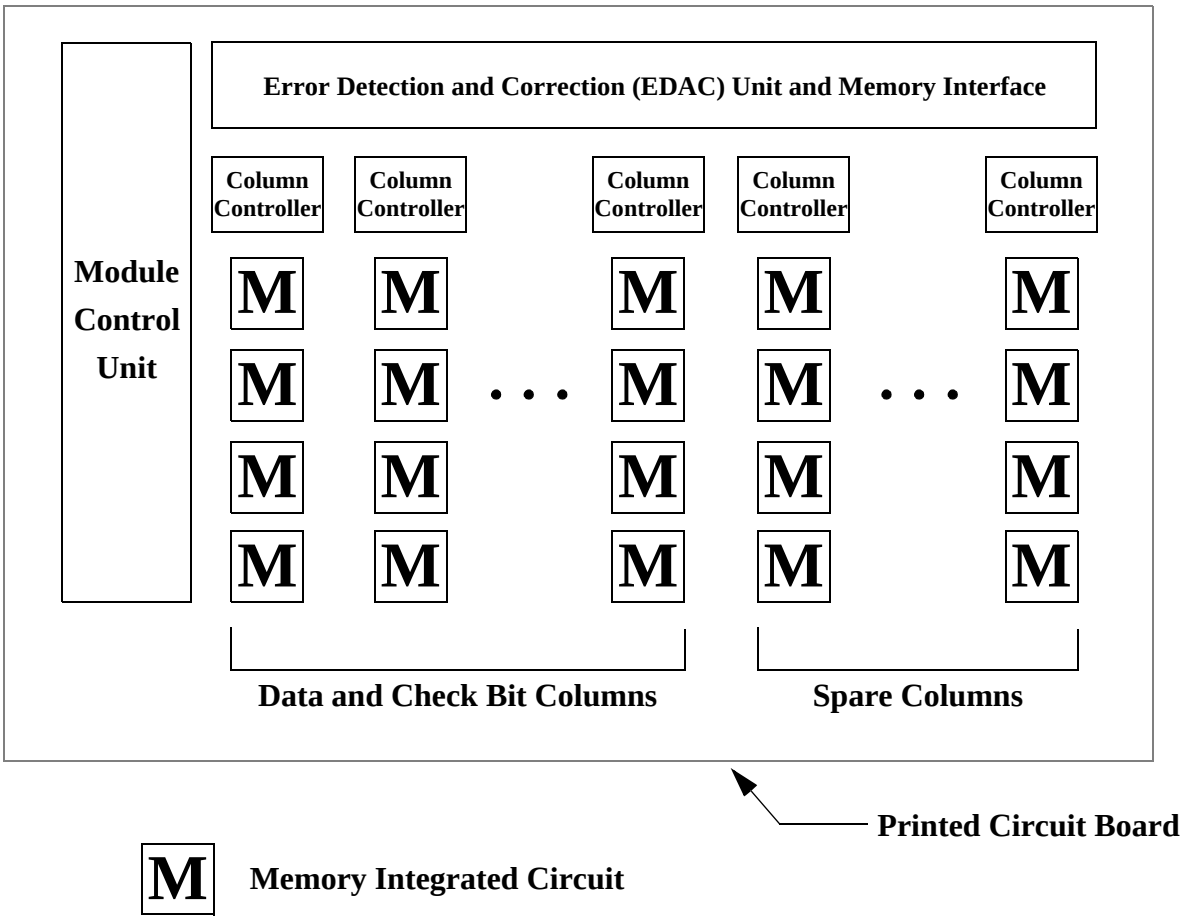



**Figure 1.23** Specific organization of a memory module using column sparing.

rate is typically denoted as $\lambda$ when it is assumed to have a constant value. The failure rate is one measure that can be used to compare systems or components.

To more clearly understand the mathematical basis for the concept of a failure rate, first recall the definition of the reliability function. The reliability, *R(t)*, of a component, or a system, is the conditional probability that the component operates correctly throughout the interval $[t_0, t]$ given that it was operating correctly at the time $t_0$. Suppose that we run a test on *N* identical components by placing all *N* components in operation at time $t_0$ and recording the number of failed and working components at time *t*. Let $N_f(t)$ be the number of components that have failed at time *t* and $N_o(t)$ be the number of components that are operating correctly at time *t*. It is assumed that once a compo-

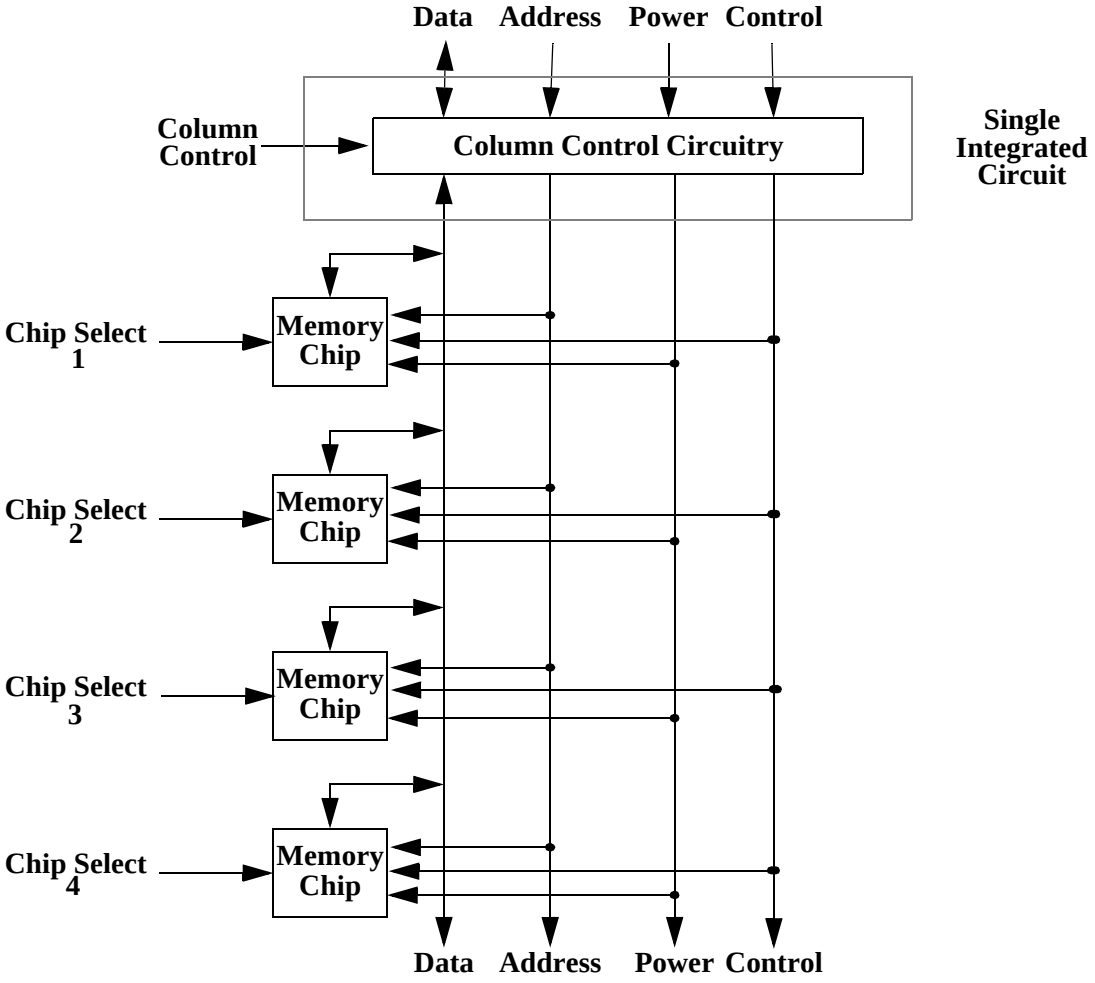



**Figure 1.24** Organization of each column within a memory module which uses column sparing.

nent fails it remains failed indefinitely. The reliability of the components at time *t* is given by

$$R(t) = \frac{N_o(t)}{N} = \frac{N_o(t)}{N_o(t) + N_f(t)}$$

which is simply the probability that a component has survived the interval $[t_o, t]$. The probability that a component has not survived the time interval is called the unreliability and is given by

$$Q(t) = \frac{N_f(t)}{N} = \frac{N_f(t)}{N_o(t) + N_f(t)}$$

Notice that at any time *t*, *R(t)*=1.0 - *Q(t)* because

$$R(t) + Q(t) = \frac{N_o(t) + N_f(t)}{N_o(t) + N_f(t)} = 1.0$$

If we write the reliability function as and differentiate *R(t)* with respect to time, we obtain

$$R(t) = 1.0 - \frac{N_f(t)}{N}$$

$$\frac{dR(t)}{dt} = \left(-\frac{1}{N}\right)\frac{dN_f(t)}{dt}$$

which can be written as

$$\frac{dN_f(t)}{dt} = (-N)\frac{dR(t)}{dt}$$

The derivative of $N_f(t)$, $\frac{dN_f(t)}{dt}$, is simply the instantaneous rate at which components are failing. At time *t*, there are still $N_o(t)$ components operational. Dividing $\frac{dN_f(t)}{dt}$ by $N_o(t)$ we obtain

$$z(t) = \frac{1}{N_o(t)}\frac{dN_f(t)}{dt}$$

*z(t)* is called the *hazard function*, *hazard rate*, or *failure rate function*. The units for the failure rate function are failures per unit of time.

There are a number of different ways in which the failure rate function can be expressed. For example, *z(t)* can be written strictly in terms of the reliability function, *R(t)*, as

$$z(t) = \frac{1}{N_o(t)}\frac{dN_f(t)}{dt} = \frac{1}{N_o(t)}\left(-N\frac{dR(t)}{dt}\right) = -\frac{\frac{dR(t)}{dt}}{R(t)}$$

Similarly, *z(t)* can be written in terms of the unreliability, *Q(t)*, as

$$z(t) = -\frac{\frac{dR(t)}{dt}}{R(t)} = \frac{\frac{dQ(t)}{dt}}{1 - Q(t)}$$

The derivative of the unreliability, $\frac{dQ(t)}{dt}$ , is called the *failure density function.*

The failure rate function is clearly dependent upon time; however, experience has shown that the failure rate function for electronic components does have a period where the value of *z(t)* is approximately constant. The commonly accepted relationship between the failure rate function and time for electronic components is called the bathtub curve and is illustrated in Figure 1.25. The bathtub curve assumes that during the early life of systems failures occur frequently due to sub-standard or weak components. The decreasing part of the bathtub curve is called the early-life or infant mortality region. At the opposite end of the curve is the wear-out region where systems have been functional for a long period of time and are beginning to experience failures due to the physical wearing of electronic or mechanical components. The increasing part of the bathtub curve is called the wear-out phase. During the intermediate region, the failure rate function is assumed to be a constant. The constant portion of the bathtub curve is called the useful life phase of the system, and the failure rate function is assumed to have a value of $\lambda$ during that period. $\lambda$ is referred to as the failure rate and is normally expressed in units of failures per hour.

The period of a constant failure rate is typically the most useful portion of a system's life. During the useful life phase, the system is providing its most predictable service to its users. We usually attempt to get a system beyond the infant mortality stage by using the concept of burn-in to remove weak components. Burn-in implies operating a system, often at an accelerated pace, prior to placing the system into service to get the system to the beginning of the useful-life period. In addition, the system is normally replaced before it enters the wear-out phase of its life. Thus, the primary interest is the performance of the system during the useful-life phase.

As noted previously, the failure rate function can be related to the reliability function as

$$z(t) = \frac{1}{N_o(t)}\frac{dN_f(t)}{dt} = -\frac{N}{N_o(t)}\frac{dR(t)}{dt} = -\frac{\frac{dR(t)}{dt}}{R(t)}$$

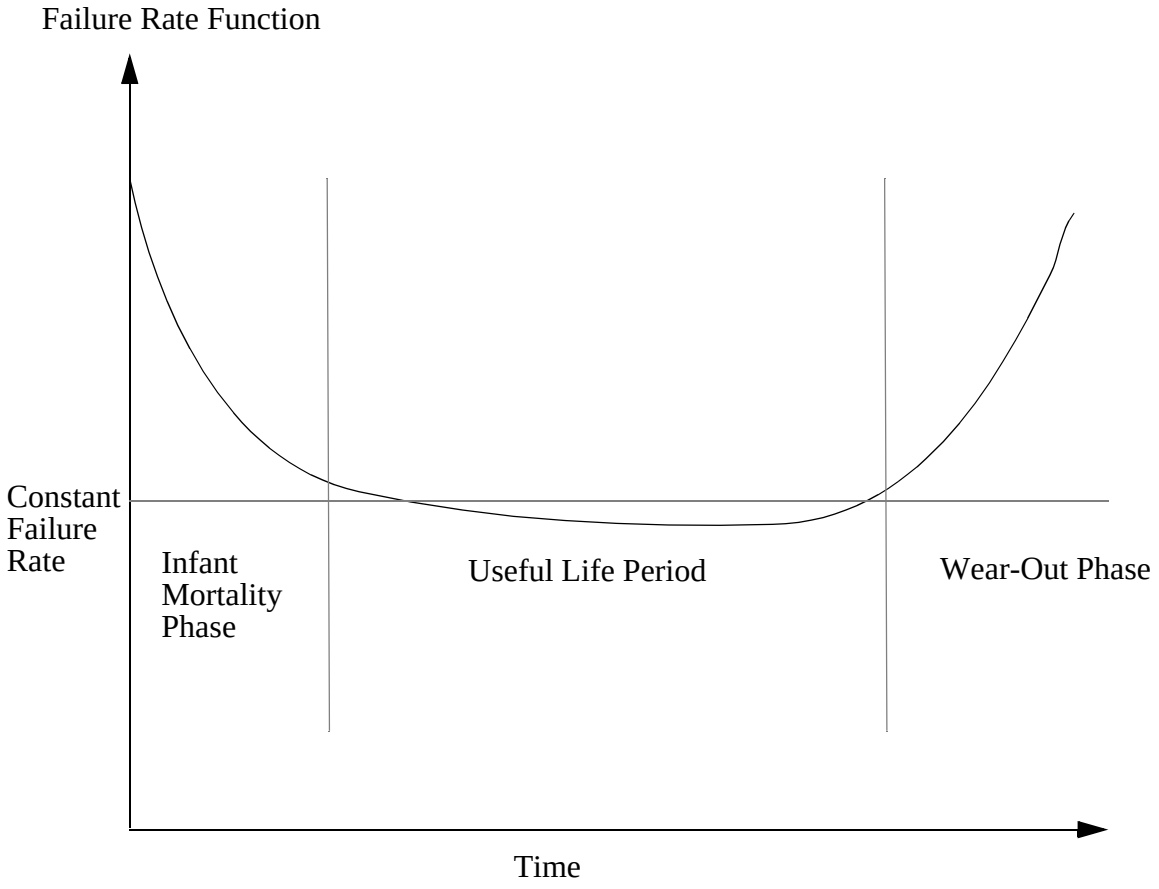



**Figure 1.25** Illustration of the bathtub curve relationship [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 173].

The result is a differential equation of the form

$$\frac{dR(t)}{dt} = -z(t)R(t)$$

The general solution of this differential equation is given by

$$R(t) = e^{-\int z(t)dt}$$

If we assume that the system is in the useful-life stage where the failure rate function has a constant value of $\lambda$, the solution to the differential equation is an exponential function of the parameter $\lambda$

given by

$$R(t) = e^{-\lambda t}$$

where $\lambda$ is the constant failure rate. The exponential relationship between the reliability and time is known as the *exponential failure law* which states that for a constant failure rate function, the reliability varies exponentially as a function of time.

The exponential failure law is extremely valuable for the analysis of electronic components and is by far the most commonly used relationship between reliability and time. Many cases, however, cannot assume that the failure rate function is constant, so the exponential failure law cannot be used; other modeling schemes and representations must be employed. An example of a time-varying failure rate function is found in the analysis of software. Software failures are the result of design faults, and as a software package is used design faults will be discovered and corrected. Consequently, the reliability of software should improve as a function of time, and the failure rate function should decrease.

A common modeling technique used to represent time-varying failure rate functions is the Weibull distribution [SIEWIOREK 82]. The failure rate function associated with the Weibull distribution is given by

$$z(t) = \alpha\lambda(\lambda t)^{\alpha - 1}$$

where $\alpha$ and $\lambda$ are constants that control the variation of the failure rate function with time. The failure rate function given by the Weibull distribution is intuitively appealing. For example, if the value of $\alpha$ is 1, *z(t)* is simply the constant $\lambda$. If $\alpha$ is greater than 1, *z(t)* will increase as time increases, and if $\alpha$ is less than 1, *z(t)* will decrease as time increases.

The reliability function that results from the Weibull distribution is the solution to the differential equation

$$\frac{dR(t)}{dt} = -z(t)R(t) = -\alpha\lambda(\lambda t)^{\alpha - 1}R(t)$$

and is given by

$$R(t) = e^{-(\lambda t)^{\alpha}}$$

The expression for *R(t)* can be verified by calculating the derivative of *R(t)*. Specifically,

$$\frac{dR(t)}{dt} = -e^{-(\lambda t)^{\alpha}}\alpha\lambda(\lambda t)^{\alpha-1} = -\alpha\lambda(\lambda t)^{\alpha-1}e^{-(\lambda t)^{\alpha}} = -z(t)R(t)$$

As stated earlier, certain values of $\alpha$ result in a reliability function that increases as time increases. For example, if $\alpha$ = -1, the reliability is given by

$$R(t) = e^{-\frac{1}{\lambda t}}$$

which approaches 1 as *t* approaches infinity and is 0 when *t* is 0. Also note that for $\alpha$ = 1, the reliability function is identical to the exponential failure law.

**Mean Time To Failure.** In addition to the failure rate, the mean time to failure (*MTTF*) is a useful parameter to specify the quality of a system. The *MTTF* is the expected time that a system will operate before the first failure occurs. For example, if we have *N* identical systems placed into operation at time *t* = 0, and we measure the time that each system operates before failing, the average time is the *MTTF*. If each system, *i*, operates for a time, $t_i$, before encountering the first failure, the *MTTF* is given by

$$MTTF = \sum_{i=1}^{N} \frac{t_i}{N}$$

The *MTTF* can be calculated by finding the expected value of the time of failure. From probability theory, we know that the expected value of a random variable, *X*, is

$$E[X] = \int_{-\infty}^{\infty} xf(x)dx$$

where *f(x)* is the probability density function. In reliability analysis we are interested in the

expected value of the time of failure (*MTTF*), so

$$MTTF = \int_0^\infty t f(t) dt$$

where *f(t)* is the failure density function, and the integral runs from 0 to ∞ because the failure density function is undefined for times less than 0. We know, however, that the failure density function is

$$f(t) = \frac{dQ(t)}{dt}$$

so, the *MTTF* can be written as

$$MTTF = \int_0^\infty t \frac{dQ(t)}{dt} dt$$

Using integration by parts and the fact that $\frac{dQ(t)}{dt} = -\frac{dR(t)}{dt}$, we can show that

$$MTTF = \int_0^\infty t \frac{dQ(t)}{dt} dt = -\int_0^\infty t \frac{dR(t)}{dt} dt = [-tR(t) + \int R(t) dt]\Big|_0^\infty = \int_0^\infty R(t) dt$$

The term *-tR(t)* clearly disappears when $t = 0$; but, it also disappears when $t = \infty$ because $R(\infty) = 0$. Consequently, the *MTTF* is defined in terms of the reliability function as

$$MTTF = \int_0^\infty R(t) dt$$

which is valid for any reliability function that satisfies $R(\infty) = 0$.

**Mean Time To Repair.** The Mean Time To Repair (*MTTR*) is simply the average time required to repair a system. The *MTTR* is extremely difficult to estimate and is often determined experimentally by injecting a set of faults, one at a time, into a system and measuring the time required to repair the system in each case. The measured repair times are averaged to determine an average time to repair. In other words, if the $i^{th}$ of *N* faults requires a time $t_i$ to repair, the *MTTR*

will be estimated as

$$MTTR = \frac{\sum_{i=1}^{N} t_i}{N}$$

The *MTTR* is normally specified in terms of a repair rate, $\mu$, which is the average number of repairs that occur per time period. The units of the repair rate are normally number of repairs per hour. The *MTTR* and the repair rate, $\mu$, are related by

$$MTTR = \frac{1}{\mu}$$

**Mean Time Between Failure.** It is very important to understand the difference between the *MTTF* and the mean time between failure (*MTBF*). Unfortunately, these two terms are often used interchangeably. While the numerical difference is small in many cases, the conceptual difference is very important. The *MTTF* is the average time until the first failure of a system, while the *MTBF* is the average time between failures of a system. As noted in the previous section, we can estimate the *MTTF* for a system by placing each of a population of *N* identical systems into operation at time $t = 0$, measuring the time required for each system to encounter its first failure, and averaging these times over the *N* systems. The *MTBF*, however, is calculated by averaging the time between failures, including any time required to repair the system and place it back into an operational status. In other words, each of the *N* systems is operated for some time *T* and the number of failures encountered by the $i^{th}$ system is recorded as $n_i$. The average number of failures is computed as

$$n_{avg} = \sum_{i=1}^{N} \frac{n_i}{N}$$

Finally, the *MTBF* is

$$MTBF = \frac{T}{n_{avg}}$$

In other words, the *MTBF* is the total operation time, *T*, divided by the average number of failures experienced during the time *T*.

If we assume that all repairs to a system make the system perfect once again just as it was when it was new, the relationship between the *MTTF* and the *MTBF* can be determined easily. Once successfully placed into operation, a system will operate, on the average, a time corresponding to the *MTTF* before encountering the first failure. The system will then require some time, *MTTR*, to repair the system and place it back into operation once again. The system will then be perfect once again and will operate for a time corresponding to the *MTTF* before encountering its next failure. The time between the two failures is the sum of the *MTTF* and the *MTTR* and is the *MTBF*. Thus, the difference between the *MTTF* and the *MTBF* is the *MTTR*. Specifically, the *MTBF* is given by

$$MTBF = MTTF + MTTR$$

In most practical applications the *MTTR* is a small fraction of the *MTTF*, so the approximation that the *MTBF* and *MTTF* are equal is often quite good. Conceptually, however, it is crucial to understand the difference between the *MTBF* and the *MTTF*.

**Fault Coverage.** An extremely important parameter in the design and analysis of fault-tolerant systems is fault coverage. The fault coverage available in a system can have a tremendous impact on the reliability, safety, and other attributes of the system. There are several types of fault coverage, depending upon whether the designer is concerned with fault detection, fault location, fault containment, or fault recovery. In addition, there are two primary definitions of fault coverage; one that is intuitive, another that is more mathematical.

The intuitive definition is that coverage is a measure of a system's ability to perform fault detection, fault location, fault containment, and/or fault recovery. The four primary types of fault coverage are fault detection coverage, fault location coverage, fault containment coverage, and fault recovery coverage. Fault detection coverage is a measure of a system's ability to detect faults. For example, a system requirement may be that a certain fraction of all faults be detected; the fault detection coverage is a measure of the system's capability to meet such a requirement. Fault location coverage is a measure of a system's ability to locate faults. Once again, it is very common to require a system to locate faults to within easily replaceable modules, and the fault location coverage is a measure of the success with which fault location is performed. Fault containment coverage

is a measure of a system's ability to contain faults Finally, fault recovery coverage is a measure of a system's ability to recover from faults and maintain an operational status. Clearly, a high fault recovery coverage will require high fault detection, location, and containment coverages.

In the evaluation of fault-tolerant systems, the fault recovery coverage is the most commonly considered, and the general term fault coverage is often used to mean fault recovery coverage. In other words, the term fault coverage is interpreted as a measure of a system's ability to successfully recover after the occurrence of a fault, therefore tolerating the fault. When using the term fault coverage, however, it is important to understand whether the coverage applies to detection, location, containment, or recovery. In the remainder of this chapter we will use the term fault coverage to imply fault recovery coverage since fault recovery is the most common form of coverage encountered. In all cases, however, it will be made clear whether detection, location, containment, or recovery coverage is being considered.

Fault coverage is mathematically defined as the conditional probability that, given the existence of a fault, the system recovers [BOURICIUS 69]. Recall that fault recovery is the process of maintaining or regaining operational status after a fault occurs. The fundamental problem with fault coverage is that it is extremely difficult to calculate. Probably the most common approach to estimating fault coverage is to develop a list all of the faults that can occur in a system and to form, from that list, a list of faults that can be detected, a list of faults that can be located, a list of faults that can be contained, and a list of faults from which the system can recover. The fault detection coverage factor, for example, is then computed as simply the fraction of faults that can be detected; that is, the number of faults detected divided by the total number of faults. The remaining fault coverage factors are calculated in a similar manner.

Several important points should be made about the estimation of coverage. First, the estimation of fault coverage requires the definition of the types of faults that can occur. Stating that the fault detection coverage is 0.9, for example, is meaningless unless the types of faults considered are identified. A second important point about the fault coverage is that it is typically assumed to be a constant. It is easy to envision applications in which the probability of detecting a fault, for example, increases as a function of time, after the occurrence of the fault. However, to simplify the analysis, the various fault coverages are normally assumed to be constants.

## 1.3.2. Reliability Modeling

Reliability is perhaps one of the most important attributes of systems. Almost all specifications for systems mandate that certain values for reliability be achieved and in some way demonstrated. The most popular reliability analysis techniques are the analytical approaches. Of the analytical techniques, combinatorial modeling and Markov modeling are the two most commonly used approaches. Here, we will consider both the combinatorial and the Markov models.

**Combinatorial Models.** Combinatorial models use probabilistic techniques that enumerate the different ways in which a system can remain operational. The probabilities of the events that lead to a system being operational are calculated to form an estimate of the system's reliability. The reliability of a system is generally derived in terms of the reliabilities of the individual components of the system. The two models of systems that are most common in practice are the series and the parallel. In a series system, each element of the system is required to operate correctly for the system to operate correctly. In a parallel system, on the other hand, only one of several elements must be operational for the system to perform its functions correctly.

The series system is best thought of as a system that contains no redundancy; that is, each element of the system is needed to make the system function correctly. In general, a system may contain N elements, and in a series system each of the N elements is required for the system to function correctly. The reliability of the series system can be calculated as the probability that none of the elements will fail. Another way to look at this is that the reliability of the series system is the probability that all of the elements are working properly.

Suppose we let $C_{iw}(t)$ represent the event that component $C_i$ is working properly at time $t$, $R_i(t)$ is the reliability of component $C_i$ at time $t$, and $R_{series}(t)$ is the reliability of the series system. Further suppose that the series system contains $N$ series components. The reliability at any time, $t$, is the probability that all $N$ components are working properly. In mathematical terms,

$$R_{series}(t) = P\{C_{1w}(t) \cap C_{2w}(t) \cap \ldots \cap C_{Nw}\}$$

Assuming that the events, $C_{iw}(t)$, are independent, we have

$$R_{series}(t) = R_1(t)R_2(t)\ldots R_N(t)$$

or

$$R_{series}(t) = \prod_{i=1}^{N} R_i(t)$$

An interesting relationship exists in a series system if each individual component satisfies the exponential failure law. Suppose that we have a series system made up of *N* components, and each component, *i*, has a constant failure rate of $\lambda_i$. Also assume that each component satisfies the exponential failure law. The reliability of the series system is given by

$$R_{series}(t) = e^{-\lambda_1 t} e^{-\lambda_2 t} \ldots e^{-\lambda_N t}$$

or

$$R_{series}(t) = e^{\sum_{i=1}^{N} \lambda_i t}$$

The distinguishing feature of the basic parallel system is that only one of *N* identical elements is required for the system to function. The unreliability of the parallel system can be computed as the probability that all of the *N* elements fail. Suppose that we let $C_{if}(t)$ represent the event that element i in the parallel system has failed at time *t*, $Q_{parallel}(t)$ be the unreliability of the parallel system, and $Q_i(t)$ be the unreliability of the $i^{th}$ element. $Q_{parallel}(t)$ can be computed as

$$Q_{parallel}(t) = P\{C_{1f}(t) \cap C_{2f}(t) \cap \ldots \cap C_{Nf}(t)\}$$

or

$$Q_{parallel}(t) = Q_1(t) Q_2(t) \ldots Q_N(t) = \prod_{i=1}^{N} Q_i(t)$$

The reliability of the parallel system can now be computed because we know that the reliability and the unreliability must add to 1.0. Mathematically, we must have $R(t) + Q(t) = 1.0$ for any

system. Consequently, we can write

$$R_{parallel}(t) = 1.0 - Q_{parallel}(t) = 1.0 - \prod_{i=1}^{N} Q_i(t) = 1.0 - \prod_{i=1}^{N} (1.0 - R_i(t))$$

It should be noted that the equations for the parallel system assume that the failures of the individual elements that make up the parallel system are independent. For random hardware failures, the independence of failures is a good assumption; however, for failures that are the result of items such as external disturbances, the independence assumption is not very good. Therefore, the combinatorial modeling techniques are most often applied to the analysis of random failures in the hardware of a system.

*M*-of-*N* systems are a generalization of the ideal parallel system. In the ideal parallel system, only one of *N* modules is required to work for the system to work. In the *M*-of-*N* system, however, *M* of the total of *N* identical modules are required to function for the system to function. A good example is the *TMR* configuration where two of the three modules must work for the majority voting mechanism to function properly. Therefore, the *TMR* system is a 2-of-3 system.

In general, if there are *N* identical modules and *M* of those are required for the system to function properly, then the system can tolerate *N*-*M* module failures. The expression for the reliability of an *M*-of-*N* system can be written as

$$R_{M-of-N}(t) = \sum_{i=0}^{N-M} \binom{N}{i} R^{N-i}(t)(1.0 - R(t))^{i}$$

where

$$\binom{N}{i} = \frac{N!}{(N-i)!\,i!}$$

**Markov Models.** The primary difficulty with the combinatorial models is that many complex systems cannot be modeled easily in a combinatorial fashion. The reliability expressions are often very complex. In addition, the fault coverage that we have seen to be extremely important in the reliability of a system is sometimes difficult to incorporate into the reliability expression in a combinatorial model. Finally, the process of repair that occurs in many systems is very difficult to

model in a combinatorial fashion. For these reasons, we often use Markov models, which are sometimes referred to as Markov chains.

The purpose of the presentation in this chapter is not to delve into the mathematical details of Markov models but to understand how to use Markov models. For the reader interested in more explicit mathematical details please refer to the references ([SHOOMAN 68] and [TRIVEDI 82]). The discussions here will provide sufficient mathematical background to apply the Markov model but will not pursue various techniques for solving the models. The presentation here is derived partially from [CHOI 92] and [JOHNSON 89].

The two main concepts in the Markov model are the state and the state transition. The state of a system represents all that must be known to describe the system at any given instant of time. For reliability models, each state of the Markov model represents a distinct combination of faulty and fault-free modules. The state transitions govern the changes of state that occur within a system. As time passes and failures and reconfigurations occur, the system will go from one state to another. The state transitions are characterized by probabilities such as the probability of failure, fault coverage, and the probability of repair.

As an example of the state transitions that can occur, consider the TMR system. The state diagram that results for the TMR system is shown in Figure 1.26. As can be seen, the system begins in state (111), and, upon the first module failure, will transition to either state (110), (101), or (011) depending upon whether module 1, 2, or 3 is the module that fails. Notice that the transition exists for the model to remain in a state if a module failure does not occur. The states in the diagram shown in Figure 1.26 can be partitioned into three major categories; the perfect state, (111), in which all modules function correctly, the one-failed states, (110), (101), and (011), in which a single module has failed, and the system-failed states, (100), (001), (010), and (000), in which enough modules have failed to cause the system to fail. State partitioning will be useful later on when we attempt to reduce the Markov model.

Each state transition has associated with it a transition probability. If we assume that each module in the TMR system obeys the exponential failure law with a constant failure rate of $\lambda$, the probability of a module being failed at some time $t + \Delta t$, given that the module was operational at

time *t*, is given by

$$1 - \frac{R(t+\Delta t)}{R(t)} = 1 - \frac{e^{-\lambda(t+\Delta t)}}{e^{-\lambda t}}$$

$$= 1 - e^{-\lambda \Delta t}$$

If we expand the exponential, we obtain

$$1 - e^{-\lambda \Delta t} = 1 - \left[1 + (-\lambda \Delta t) + \frac{(-\lambda \Delta t)^2}{2!} + \ldots\right] = (-\lambda \Delta t) - \left(\frac{(-\lambda \Delta t)^2}{2!}\right) - \ldots$$

For values of $\Delta t$ such that $\lambda \Delta t \ll 1$, the expression reduces to

$$1 - e^{-\lambda \Delta t} \approx \lambda \Delta t$$

Thus, using the exponential failure law allows us to state that the probability that a component will fail within the time period $\Delta t$ given that the component was operational at time *t* is approximately $\lambda \Delta t$. When states are "collapsed", the transition probability to the resulting state is the sum

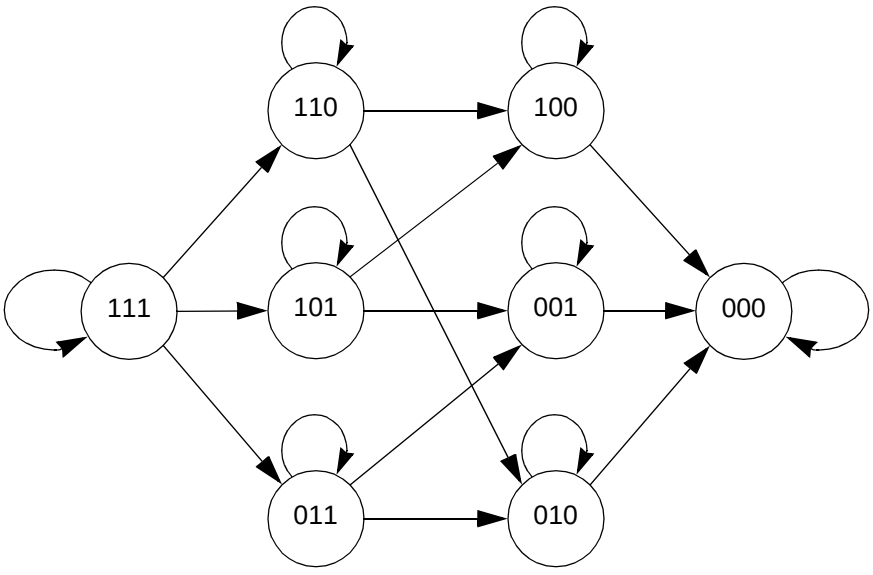



**Figure 1.26** Markov model of a TMR system [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 201].

of all the transitions from an arbitrary state to the individual categorized states. This is illustrated in Figure 1.27 and is usually referred to as a reduced or collapsed Markov chain. The advantage of state reduction is the lower cost in computation; instead of evaluating an eight-state Markov chain, through state reduction we now only have to evaluate a three-state Markov chain.

Examining the TMR system further, we can now specify the transition probabilities as shown in Figure 1.27. Note that the state space shown can be partitioned into three categories: a perfect state, a one-module-failed state, and a system-failed state. In other words, by revising the state transition probabilities appropriately, we have collapsed the enumerated TMR state space to these three states.

With the revised Markov chain of a TMR system, let state 3 correspond to the perfect state (111), state 2 correspond to the grouping of one-failed states (110), (101), (011) and state *F* correspond to the group of states with 2 modules failed and the completely failed state. The transition probabilities are derived to account for one of several failures occurring. Consequently the transition from state 3 to state 2 is the sum of all transitions from the perfect state to the one-failed state, $3\lambda\Delta t$. In other words, $3\lambda\Delta t$ is the probability that one out of three of the components will fail. The transition from state 2 to state F is the sum of the transitions from a one-module-failed state to a two-module failed state, $2\lambda\Delta t$.

Once the transitions for a Markov chain have been determined, extracting a state transition matrix from the chain is straightforward. Given the Markov property, we can state that the probability of being in any given state *s* at some time $t + \Delta t$ depends on the probability that the system

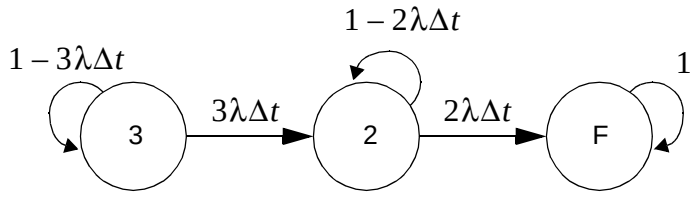



**Figure 1.27** Reduced Markov model of a TMR system [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 203].

was in a state from which it could transition to state *s* and the probability of that transition occurring. For example, the probability that the TMR system will be in state 3 at time $t + \Delta t$ depends on the probability that the system was in state 3 at time *t and* the probability of the system transitioning from state 3 back into state 3. In other words,

$$p_3(t + \Delta t) = (1 - 3\lambda\Delta t)p_3(t)$$

where $p_3(t)$ is the probability of being in state 3 at time *t* and $p_3(t + \Delta t)$ is the probability of being in state 3 at time $t + \Delta t$. Similarly, the remaining equations for states 2 and F are

$$p_2(t + \Delta t) = 3\lambda\Delta t p_3(t) + (1 - 2\lambda\Delta t)p_2(t)$$

$$p_F(t + \Delta t) = 2\lambda\Delta t p_2(t) + p_F(t)$$

Likewise definitions for the $p_2(t + \Delta t)$, $p_2(t)$, and $p_F(t + \Delta t)$, $p_F(t)$ exist.

The three equations $p_3(t + \Delta t)$, $p_2(t + \Delta t)$, $p_F(t + \Delta t)$, can be written in matrix form as

$$\begin{bmatrix} p_3(t + \Delta t) \\ p_2(t + \Delta t) \\ p_F(t + \Delta t) \end{bmatrix} = \begin{bmatrix} 1 - 3\lambda\Delta t & 0 & 0 \\ 3\lambda\Delta t & 1 - 2\lambda\Delta t & 0 \\ 0 & 2\lambda\Delta t & 1 \end{bmatrix} \begin{bmatrix} p_3(t) \\ p_2(t) \\ p_F(t) \end{bmatrix}$$

In a more compact manner this is written as

$$\mathbf{P}(t + \Delta t) = \mathbf{A}\mathbf{P}(t)$$

where

$$\mathbf{P}(t + \Delta t) = \begin{bmatrix} p_3(t + \Delta t) \\ p_2(t + \Delta t) \\ p_F(t + \Delta t) \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 1 - 3\lambda\Delta t & 0 & 0 \\ 3\lambda\Delta t & 1 - 2\lambda\Delta t & 0 \\ 0 & 2\lambda\Delta t & 1 \end{bmatrix} \quad \mathbf{P}(t) = \begin{bmatrix} p_3(t) \\ p_2(t) \\ p_F(t) \end{bmatrix}$$

and $\mathbf{P}(t)$ is the probability state vector at time *t*, $\mathbf{P}(t + \Delta t)$ is the probability state vector at time $t + \Delta t$ and $\mathbf{A}$ is the transition matrix. Note that the above equation can be viewed as a difference equation. If we set $t = 0$ ,then we have $\mathbf{P}(\Delta t) = \mathbf{A}\mathbf{P}(0)$. Similarly, for time $2\Delta t$, $\mathbf{P}(2\Delta t) = \mathbf{A}\mathbf{P}(\Delta t) = \mathbf{A}^2\mathbf{P}(0)$ . Extending this result to $n\Delta t$ yields

$$\mathbf{P}(n\Delta t) = \mathbf{A}^n\mathbf{P}(0)$$

It is also possible to obtain closed-form solutions to the Markov models. The above set of equations can be manipulated algebraicly to form

$$\frac{p_3(t+\Delta t) - p_3(t)}{\Delta t} = -3\lambda p_3(t)$$

$$\frac{p_2(t+\Delta t) - p_2(t)}{\Delta t} = 3\lambda p_3(t) - 2\lambda p_2(t)$$

$$\frac{p_F(t+\Delta t) - p_F(t)}{\Delta t} = 2\lambda p_2(t)$$

Taking the limit as $\Delta t$ approaches zero results in a set of differential equations given by

$$\frac{dp_3(t)}{dt} = -3\lambda p_3(t)$$

$$\frac{dp_2(t)}{dt} = 3\lambda p_3(t) - 2\lambda p_2(t)$$

$$\frac{dp_F(t)}{dt} = 2\lambda p_2(t)$$

The solution to this set of differential equations is

$$p_3(t) = e^{-3\lambda t}$$

$$p_2(t) = 3e^{-2\lambda t} - 3e^{-3\lambda t}$$

$$p_F(t) = 1 - 3e^{-2\lambda t} + 2e^{-3\lambda t}$$

Note that the reliability is the sum of the probabilities of being in states 3 and 2, or conversely, the reliability is 1.0 minus the probability of being in the failed state.

### 1.3.3. Safety Modeling

As mentioned before, the prime benefit of using Markov chains in dependability modeling is the ability to handle coverage in a systematic fashion. Coverage is the probability that a fault will be handled correctly given that the fault has occurred; it is usually denoted as *C*. The incorporation of

coverage in a Markov chain analysis of a system allows us to model the safety of a system because it accounts for the effectiveness of a fault coverage strategy. Previously, we only considered systems that had perfect coverage, where $C = 1.0$. Now we will look at the case where $0 \le C < 1.0$.

The implication of incorporating coverage in a Markov chain is that now, every unfailed state in the state space has two transition paths to two different states; one of which is covered, and the other is uncovered. This implies that given *n* states in a fully covered system, for an identical system that is not fully covered, there can be a maximum of $2^n$ - 1 states in the system's state space. Figure 1.28 illustrates this case for a simplex (one component) system. For a simplex system, there are three states, a fully operational state *O*, a failed-safe state FS, and a failed-unsafe state *FU*. Converting this graph to a transition matrix we get

$$\begin{bmatrix} p_O(t+\Delta t) \\ p_{FS}(t+\Delta t) \\ p_{FU}(t+\Delta t) \end{bmatrix} = \begin{bmatrix} 1-\lambda\Delta t & 0 & 0 \\ \lambda\Delta t C & 1 & 0 \\ \lambda\Delta t(1-C) & 0 & 1 \end{bmatrix} \begin{bmatrix} p_O(t) \\ p_{FS}(t) \\ p_{FU}(t) \end{bmatrix}$$

The safety of the system described by the Markov model of Figure 1.28 can be written as

$$S(t) = p_O(t) + p_{FS}(t)$$

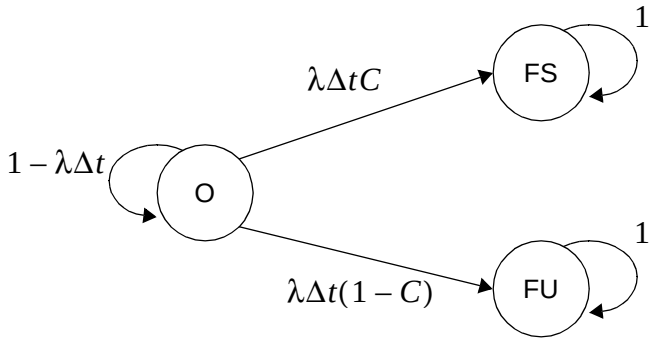



**Figure 1.28** Markov model of a simplex system with coverage [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 214].

where $S(t)$ is the safety, $p_O(t)$ is the probability of being in the operational state at time $t$, and $p_{FS}(t)$ is the probability of being in the failed-safe state at time $t$. The time domain solutions to this set of equations can be written as

$$p_{FS}(t) = C - Ce^{-\lambda t}$$

$$p_{FU}(t) = (1 - C) - (1 - C)e^{-\lambda t}$$

$$p_O(t) = e^{-\lambda t}$$

The safety of the system can now be written as

$$S(t) = p_O(t) + p_{FS}(t) = C + (1 - C)e^{-\lambda t}$$

At time zero, the safety of the system is 1.0, as expected. As time approaches infinity, however, the safety will approach a steady-state value known as the steady-state safety of the system. Specifically, we find that

$$S(\infty) = C$$

In other words, the steady-state safety depends directly on the fault coverage provided by the system.

### 1.3.4. Availability Modeling

Now consider a system that incorporates repair. For all of the Markov chains previously described, each chain had the property of being acyclic. Introducing the notion of repair implies being able to return from a less operational or failed state to a state that is more operational or fully operational. As such, all that needs to be done to incorporate repair in a Markov chain is to specify a return transition probability. Assume that the system has a constant repair rate $\mu$. Given that the repair rate is analogous to the failure rate in that it represents the number of repairs expected to occur within a certain time period, the probability of repair occuring within a certain time period $\Delta t$ given that the system was failed or less operational at time t would be $\mu\Delta t$. To illustrate repair, consider the example of a fully covered simplex system with repair. Figure 1.29 shows the enu-

merated state space of such a system. The Markov equations for this system are

$$\begin{bmatrix} p_O(t+\Delta t) \\ p_F(t+\Delta t) \end{bmatrix} = \begin{bmatrix} 1-\lambda\Delta t & \mu\Delta t \\ \lambda\Delta t & 1-\mu\Delta t \end{bmatrix} \begin{bmatrix} p_O(t) \\ p_F(t) \end{bmatrix}$$

Extraction of an availability calculation is identical in method to the reliability calculation, with one difference; for a reliability model there can be no repair from a nonoperational state. Availability permits the model to have a cyclic Markov chain, thus allowing one to examine the effect of repair on system performance. For this simplex example, the availability $A(t) = p_O(t)$.

The differential equations describing the model of Figure 1.29 are

$$\frac{dp_F(t)}{dt} = \lambda p_O(t) - \mu p_F(t)$$

$$\frac{dp_O(t)}{dt} = -\lambda p_O(t) + \mu p_F(t)$$

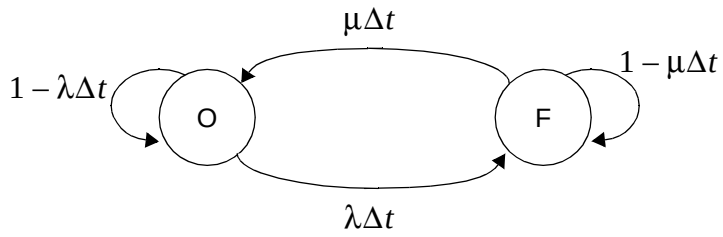



**Figure 1.29** Markov model illustrating the concept of repair [From Barry W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989, page 222].

Assuming that the model initially starts in state O, the solutions to the above equations are

$$p_O(t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t}$$

$$p_F(t) = \frac{\lambda}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t}$$

Several interesting features are apparent in the above solutions. For example, as time approaches infinity, the probability of being in the operational state approaches a steady-state value given by

$$p_O(\infty) = \frac{\mu}{\lambda + \mu} = \frac{1}{\frac{\lambda}{\mu} + 1}$$

Subsequently, we will see that this constant is known as the steady-state availability.

Recall that the availability, *A(t)*, of a system is defined as the probability that a system will be available to perform its tasks at the instant of time *t*. Intuitively, we can see that the availability can be approximated as the total time that a system has been operational divided by the total time elapsed since the system was initially placed into operation. In other words, the availability is the percentage of time that the system is available to perform its expected tasks. Suppose that we place a system into operation at time $t = 0$. As time moves along, the system will perform its functions, perhaps fail, and hopefully be repaired. At some time $t = t_{current}$, suppose that the system has operated correctly for a total of $t_{op}$ hours and has been in the process of repair or waiting for repair to begin for a total of $t_{repair}$ hours. The time $t_{current}$ is then the sum of $t_{op}$ and $t_{repair}$. The availability can be determined as

$$A(t_{current}) = \frac{t_{op}}{t_{op} + t_{repair}}$$

where $A(t_{current})$ is the availability at time $t_{current}$.

The above expression lends itself well to the experimental evaluation of the availability of a system; we can simply place the system into operation and measure the appropriate times required

to calculate the availability of the system at a number of points in time. Unfortunately, the experimental evaluation of the availability is often not possible because of the time and expense involved. Also, we would like to have some means of estimating the availability before we actually build the system so that availability considerations can be factored into the design process. One approach is based on the single parameter measures such as *MTTF* and *MTTR* and yields what is typically called the steady-state availability, $A_{ss}$.

We have seen that availability is basically the percentage of time that a system is operational. Using knowledge of the statistical interpretation of the *MTTF* and the *MTTR*, we expect that, on the average, a system will operate for *MTTF* hours and then encounter its first failure. Once the failure has occurred, the system will then, again on the average, require *MTTR* hours to be repaired and placed into operation once again. The system will then operate for another *MTTF* hours before encountering its second failure.

If the average system experiences *N* failures during its lifetime, the total time that the system will be operational is *N(MTTF)* hours. Likewise, the total time that the system is down for repairs is *N(MTTR)* hours. In other words, the operational time, $t_{op,}$ is *N(MTTF)* hours and the down-time, $t_{repair}$, is *N(MTTR)* hours. The average, or steady-state, availability is

$$A_{ss} = \frac{N(MTTF)}{N(MTTF) + N(MTTR)}$$

We know, however, that the *MTTF* and the *MTTR* are related to the failure rate and the repair rate, respectively for simplex systems, as

$$MTTF = \frac{1}{\lambda}$$

$$MTTR = \frac{1}{\mu}$$

Therefore, the steady-state availability is given by

$$A_{ss} = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda} + \frac{1}{\mu}} = \frac{1}{1 + \frac{\lambda}{\mu}}$$

Recall that the repair rate is expressed in repairs per hour while the failure rate is in failures per hour. One would expect that if the failure rate goes to zero, implying that the system never fails, or the repair rate goes to infinity, implying that no time is required to repair the system, the availability will go to 1. Looking at the expression for the steady-state availability, we can see that this is true.

### 1.3.5. Maintainability Modeling

As defined previously, the maintainability is the probability that a failed system will be restored to working order within a specified time. We will use the notation that *M(t)* is the maintainability for time *t*. In other words, *M(t)* is the probability that a system will be repaired in a time less than or equal to *t*.

An important parameter in maintainability modeling is the repair rate, $\mu$. The repair rate is the average number of repairs that can be performed per time unit. The inverse of the repair rate is the *MTTR* which is the average time required to perform a single repair. Mathematically, the relationship between the repair rate and the *MTTR* is given by

$$MTTR = \frac{1}{\mu}$$

An expression for the maintainability of a system can be derived in a manner similar to that used to develop the exponential failure law for the reliability function. Suppose that we have *N* systems, we inject one unique fault into each of the systems, and we allow one maintenance person to repair each system. We begin this experiment by injecting the faults into the systems at time *t*=0. Later, at some time *t*, we determine that $N_r(t)$ of the systems have been repaired and $N_{nr}(t)$ have not been repaired. Since the maintainability of a system at time *t* is the probability that the system

can be repaired by time *t*, an estimate of the maintainability can be computed as

$$M(t) = \frac{N_r(t)}{N} = \frac{N_r(t)}{N_r(t) + N_{nr}(t)}$$

If we differentiate *M(t)* with respect to time, we obtain

$$\frac{dM(t)}{dt} = \frac{1}{N}\frac{dN_r(t)}{dt}$$

which can also be written as

$$\frac{dN_r(t)}{dt} = N\frac{dM(t)}{dt}$$

The derivative of $N_r(t)$ is simply the rate at which components are repaired at the instant of time *t*.

At time *t*, we have $N_{nr}(t)$ systems that have not been repaired. If we divide $\frac{dN_r(t)}{dt}$ by $N_{nr}(t)$ we obtain

$$\frac{1}{N_{nr}(t)}\frac{dN_r(t)}{dt}$$

which is called the repair rate function and is assumed to have a constant value of $\mu$, the repair rate.

Using the expression for the repair rate and the expression for the derivative of $N_r(t)$, we can write

$$\mu = \frac{1}{N_{nr}(t)}\frac{dN_r(t)}{dt} = \frac{N}{N_{nr}(t)}\frac{dM(t)}{dt}$$

which yields a differential equation of the form

$$\frac{dM(t)}{dt} = \mu\frac{N_{nr}(t)}{N}$$

We know, however, that $\frac{N_{nr}(t)}{N}$ is 1 - *M(t)*, so we can write

$$\frac{dM(t)}{dt} = \mu(1 - M(t))$$

The solution to the differential equation is well known and is given by

$$M(t) = 1 - e^{-\mu t}$$

The relationship developed for *M(t)* has the desired characteristics. First, if the repair rate is zero, the maintainability will also be zero since the system cannot be repaired in any length of time. Second, if the repair rate is infinite, the maintainability will be 1.0 since repair can be performed in zero time.

The repair rate is typically specified for several levels of repair. The most common partitioning is to provide three levels of repair. The first is called the organizational level and consists of all repairs that can be performed at the site where the system is located. Organizational repairs typically include all faults that can be located to specific circuit cards such that the cards can simply be replaced and the system made operational once again. For example, if an aircraft can be repaired without bringing it off the runway, it is considered an organizational level repair. The key to organizational repairs is the ability to locate the fault. It is seldom feasible to bring sophisticated fault detection and location equipment to the site of the system. Repairs at the organizational level must often depend on the built-in-test provided by the system to locate the specific problem.

The second level of repair is called the intermediate level. Intermediate level repairs cannot be performed at the organizational level but can be performed in the immediate vicinity of the system. For example, a computer firm can have a local repair facility to which the faulty pieces of equipment are taken for repair. Intermediate level repair is not as good as being able to perform the repair on site but it is better than having to return a piece of equipment to the factory. In the case of an airplane, for example, an intermediate level repair might be made in the hanger as opposed to on the runway.

The final level of repair is called the depot level or the factory level. In depot level repairs,

the equipment must be returned to a major facility for the repair process. For example, if a calculator cannot be repaired at home (organizational level), then it is taken to the store from which it was purchased (intermediate level). If the store is unable to perform the repair, they will send it to a site designated by the manufacturer as a major repair facility (depot level). The length of time required to perform the repair depends upon the level at which it is performed. It may take less than an hour to repair a device at the organizational level, several hours or perhaps days at the intermediate level, and as much as several weeks or months at the depot level.

## 1.4. Design Methodology

The design process consists of many phases, each of which may be performed numerous times before an acceptable solution is obtained [JOHNSON 87]. The fact that the design process is iterative is extremely important. The process of performing tradeoffs will naturally result in modifications during the design process. For example, the derivation of the requirements can cause us to revisit the definition of the problem, and perhaps eliminate certain aspects of the design to make the requirements more reasonable.

### 1.4.1. The Design Process

The five primary phases of the design process are the requirements phase, the conceptual phase, the specifications phase, the design phase, and the test phase. The five phases are necessary to develop one or more prototypes of a design; we will not consider in this development the phases necessary to arrive at a manufacturable system. Embedded within the five phases are the primary steps of the design process, including problem definition, requirements determination, partitioning, candidate designs, high-level analysis, hardware and software specifications, hardware and software design, detailed analysis, testing, system integration, and system testing. Each step of the design process is briefly described in the paragraphs that follow.

**Problem Definition.** All designs clearly begin with the existence of a problem. For example, the problem may be to develop a flight-control system for a helicopter, an attitude-control system for a satellite, a network of computers to handle the transactions processing of a large bank, or a system to accurately control a nuclear reaction in a power generation plant. The first step in any

design process is to develop a description of the problem. In many cases, the ability to write on paper a clear and concise statement of the problem is a major step towards the problem's solution.

**System Requirements.** The second step in the design process is to extract, or create, a set of requirements from the problem description. The system requirements will typically consist of requirements on reliability, cost, weight, power consumption, physical size, performance (for example, speed), maintainability, and other system attributes. Many of the requirements, such as reliability, will be quantitative in nature while others, such as compatibility with existing designs, will be more qualitative features. The derivation of the requirements may cause us to revisit the problem description as we learn more about the problem and its specific attributes. Also, the definition of the problem can make certain requirements impossible to attain.

**System Partitioning.** Once the system requirements are well-defined and understood, the design process requires partitioning the problem into manageable subproblems, called subsystems, that can each be handled easily by individual teams of design engineers. For small projects, each team may be a single engineer, or one designer may handle several portions of the total design. In more complicated systems, each design team may consist of tens or hundreds of engineers. The objective of the partitioning process is to divide the complete problem into manageable pieces. The partitioning can be in terms of hardware and software or perhaps a specific division of hardware subsystems. One form of partitioning can be performed based on the system requirements.

A portion of the partitioning process involves categorizing various parts of the system based on the reliability (or availability, maintainability, or some other attribute) requirements. For illustrative purposes, we will consider the reliability. One way to partition a system based on reliability requirements is into categories of varying degrees of criticality [JOHNSON 85]. For example, the aerospace industry classifies functions as either flight-critical, mission-critical, or convenience functions. Flight-critical functions are those functions that, if discontinued or performed incorrectly, could result in the loss of the aircraft or the crew. In simple terms, a flight-critical function is one that is required to keep the aircraft flying. For example, the flight-control system in a fly-by-wire airplane is a flight-critical function. Flight-critical functions are the most important functions and, as a result, usually require the highest reliability.

Mission-critical functions are those functions that are required for the aircraft and its crew to complete its intended mission. For example, an airplane could certainly fly if the radio were to fail, but it is highly unlikely that the crew could complete its intended job. The mission-critical functions usually follow immediately the flight-critical functions in the level of reliability required.

Convenience functions are those functions that are nice to have, but that have little impact if discontinued. Compared to the flight-critical and mission-critical functions, the convenience functions, if discontinued, will produce relatively minor impacts. The electronic maintenance log in a military aircraft is another example of a convenience function. While the maintenance log is convenient and significantly improves the maintenance process, neither the crew, the mission, nor the aircraft will be endangered during flight if the maintenance log becomes inoperative.

**Concept Development.** One of the most important steps in the design of a fault-tolerant system is the creation of several candidate designs. Certainly, the initial candidates will be significantly deficient in the detail required of a complete design, but they will illustrate a basic approach that can be taken. For example, you may consider triple-modular redundancy (TMR) as a candidate approach. You may not have yet defined the mechanism used to perform the voting, the manner in which synchronization is achieved, or even the degree of synchronization required, but the basic features of TMR can be analyzed, in depth, to determine if they are appropriate for the particular application. TMR can be contrasted with standby sparing or the triple-duplex approach to determine, at least initially, which approach, if any, is best for the application at hand. One key reason for developing several candidate designs is that the process of determining the advantages of one approach will very likely uncover the disadvantages of another approach and vice versa. The analysis of the candidate designs is a key step in the overall design process.

**High-level Analysis.** Once candidate systems are defined, the next step in the design process is to perform a preliminary analysis of each candidate architecture. A preliminary analysis can consist of a reliability estimate, cost estimates, weight estimates, and so on. At this stage of the design, many candidates that are obviously not suitable for the particular application can be eliminated quickly from consideration and further detailed development. A good, high-level analysis can easily save the designers much expense by significantly decreasing the number of candidates. For example, you might initially consider TMR to be a viable alternative for an industrial control sys-

tem, but you may eliminate TMR from consideration when a high-level analysis indicates that TMR significantly exceeds the reliability requirements and the weight limitations.

The importance of some type of high-level analysis cannot be over-emphasized. Analysis, in general, must be an integral part of the design process if the design is to be successful. The earlier that deficiencies in a candidate design are identified, the more cost-effective the design process will be. You do not want a candidate to remain in consideration if significant problems exist in the approach. You also do not want to spend large amounts of time further developing a candidate that clearly cannot meet the requirements.

The difficulty with performing a high-level analysis is that many of the analysis techniques require a substantial amount of information on the specific design before an analysis can be performed. For example, we need some idea of the failure rate of the system's modules before we can determine the system's reliability [JOHNSON 84]. Likewise, if we want to simulate the system as a means of functional evaluation, many of today's simulation tools require that significant detail be available before the simulation can be constructed [BREUER 76]. In many instances, we overcome these problems by analyzing the system for a range of parameters. For example, we can calculate the reliability as a function of the failure rate and determine the reliability for a range of failure rates.

**Hardware and Software Specifications.** The term specification is often used interchangeably with the term requirement. In this chapter, however, a requirement is considered to be some attribute, or quality, that is demanded of a system. A specification, on the other hand, is a detailed plan for a design that is capable of meeting certain requirements. For example, the requirement might be to achieve a reliability of 0.995 for a simple digital filter. The specification would be the outline of the design that could meet the requirement.

Once a high-level solution to the problem has been developed, analyzed, and refined, the specifications for the hardware and software to implement the design must be developed. At this point in the design process, it is crucial that much interaction occur between the systems engineers that created the high-level design and the hardware and software design engineers that will actually create the designs. The specifications must achieve a delicate balance necessary to meet the system

requirements and, at the same time produce a practical, implementable, and manageable design.

**Hardware and Software Design and Analysis.** The result of the high-level analysis should be the selection of a final set of candidate solutions to the original problem. The set of possible solutions can contain only one candidate if all others have been eliminated for one reason or another. On the other hand, the set can contain two or more solutions that will each be carried through the complete design process to allow more detailed comparisons to be conducted. Each candidate that remains after the high-level analysis will be carried through a complete hardware and software design and construction process resulting in both hardware and software prototypes. In most instances, the hardware and software design can be performed in parallel with close interaction between the two groups to assure that hardware decisions do not negatively impact the software, and, likewise, software decisions do not negatively impact the hardware design.

It is very important to continue the analysis of the designs in parallel with the actual design and construction to assure that the system requirements (reliability, availability, maintainability, cost, and so on) are not compromised by design decisions that are made. Many times the analysis portion of the design process is overlooked until the design is complete. At that point, it is often too late to make substantial changes to remedy any design problems. As the hardware is developed, more specific data can be provided to the analysis; for example, the failure rate information and the fault coverage data. Therefore, the analysis can be refined as the design is refined. Once the design is complete, the analysis should also be complete.

**Testing.** An extremely important part of the design process is the development of a plan for testing the resulting designs and the actual testing itself. During the design process it is mandatory that testing be considered such that a testable design is conceived. Testing involves searching for faults of all types, including faults resulting from design mistakes, implementation mistakes, and component defects. The overall purpose of the test phase of the design process is to assure the correct operation of the system.

**System Integration and Test.** Once the hardware and software prototypes have been tested adequately, they must be combined to form a complete, operational system. The process of combining the hardware and the software is usually called system integration. The fundamental pur-

pose of system integration is to get all of the subsystems working together to perform the desired functions of the system. Each of the subsystems can work perfectly when tested independently of the remainder of the system; however, when all of the subsystems are required to coordinate, interfacing problems can arise. Once the system integration is complete, the system must be tested. Software faults, for example, that never create errors in the emulators used to develop the software can suddenly produce erroneous results because of hardware ideosyncracies. Likewise, hardware faults can emerge because the software is exercising the hardware in a manner slightly different than expected.

### 1.4.2. Fault Avoidance in the Design Process

The basic goal of fault avoidance is to prevent the occurrence of items that cause faults. For example, fault avoidance can be used to prevent design mistakes or implementation mistakes. The underlying theme of fault avoidance is that you do not achieve either fault tolerance, high reliability, high availability, or any other key, system attribute unless you do something during the design process to assure that the desired attributes are being designed into the system. In other words, simply testing for quality at the end of the design process is insufficient. We must incorporate procedures at all phases of the design to guarantee the achievement of a system's requirements. There are several fault avoidance techniques that can be applied at different points during the design process. Examples include various types of design reviews, adherence to design rules, shielding against external disturbances, and quality control checks.

**Requirements Design Review.** The purpose of the requirements design review is to have an independent team of design engineers review and concur with the requirements that have been derived as part of the problem definition and partitioning. It is important at this point to assure that the requirements are reasonable and will be verifiable during the analysis and testing process. It is also important to assure that all necessary requirements have been identified. The requirements design review helps eliminate specification mistakes as a cause of faults.

**Conceptual Design Review.** The conceptual design review occurs immediately after a high-level, candidate design has been developed. The purpose of the conceptual design review is to assure that the basic concept of the candidate design is correct and meets the requirements that have

been developed. During the conceptual design review, the design engineers will normally present the results of their initial analysis to justify their belief that the candidate design is capable of fulfilling all of the system requirements. In addition, the conceptual design review will examine candidates that were eliminated from consideration to verify that the reason for elimination is sufficient. The conceptual design review attempts to eliminate specification mistakes as a cause of faults by assuring that the concept that is ultimately specified is correct. The review should be performed by a team of independent designers to guarantee that the reviewers are not biased prematurely, in any way, towards, or against, the proposed, candidate solution.

**Specifications Design Review.** The result of the specifications phase of the design process is a detailed hardware and software specification and a plan for testing the resulting hardware and software. The specifications design review attempts to verify the specifications to assure their validity. Before the hardware and software designs are begun, it is important to make sure that the specifications are both correct and understood completely. Otherwise, faults due to specification mistakes will inevitably occur in the design. Also, it is extremely important to guarantee that the designs will be testable. As in the other design reviews, the specifications review should be conducted by an independent team of designers capable of completely understanding and critiqueing the material. After the completion of the review and any resulting modifications, the specifications for the hardware and software should be ready to pass along to the designers, and the detailed design process begun.

**Detailed Design Review.** Detailed design reviews must be performed on both the hardware and software designs before beginning the actual implementations. The purpose of the review is to assure that the specific designs meet the specifications and are capable of fulfilling the system requirements. A detailed design review is one of the most important and difficult reviews that must be performed because the specific details of the design must be reviewed and verified. The detailed design review must encompass both the detailed design and the detailed analysis. For example, once the detailed design is performed, specific failure rates should be known such that the accuracy of the reliability analysis can be refined.

**Final Review.** The final review is intended to be a last checkpoint in the design process. At the time of the final review, a working prototype is often available. The basic purpose of the final

review is to examine the performance of the prototype and the results of the final analysis to assure that specifications have been adhered to and the system requirements have been met. If the design has been performed correctly and the design reviews have been successful throughout the design process, the final review should not uncover any major problems.

**Parts Selection.** The selection of parts for a system can be critical to the achievement of reliability, availability, or some other system attribute. There are many levels of quality in the components that are available for use in designs; in many cases, there is a factor of 300 difference in the failure rate of a high-quality part and that same part at a lower quality. As might be expected, however, there is also a tremendous difference in the cost of parts of different qualitites. Therefore, it is extremely important to guarantee that the components are selected appropriately. The primary tradeoffs in the parts selection are: (1) the cost of the part versus the failure rate of the part, (2) availability of the part versus the failure rate, and (3) the cost (both financial and otherwise) of a failure of the part.

**Design Rules.** Design rules often play an extremely important role in the design of a system; the strict adherence to design rules can improve substantially the system. Design rules can address items such as packaging, testing, shielding, or circuit layout. For example, a design rule may require that the system be partitioned into subsystems that are no larger than some predefined value. The size of the subsystem can be controlled in terms of the number of logic gates, the physical size of the subsystem, the total number of cards required or some other metric. By partitioning based upon size, each subsystem becomes more manageable, and the probability of design mistakes is, hopefully, significantly decreased.

**Documentation.** Documentation is another example of an extremely important aspect of fault avoidance. Design projects often require that different phases of the design be handled by different teams of designers. For example, a system's architecture can be developed by a team of so-called systems engineers. Next, a team of design engineers will take the specifications developed by the systems engineers and design the hardware and software. If the top-level architecture is incorrectly documented by the systems engineers, the final hardware and software designs can be incorrect. The design reviews mentioned previously are mechanisms that can be used to detect the existence of documentation errors. It is critical to the success of a project that each stage of the design be

described correctly and clearly in the documentation.

## 1.5. References


[AVIZIENIS 71a] Avizienis, A., "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1322-1331.

[AVIZIENIS 71b] Avizienis, A., et al., "The STAR (Self-Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Transactions on Computers*, Vol. C-20, No. 11, November 1971, pp. 1312-1321.

[AVIZIENIS 82] Avizienis, A., "The Four-Universe Information System Model for the Study of Fault Tolerance," *Proceedings of the 12th Annual International Symposium on Fault-Tolerant Computing*, Santa Monica, California, June 22-24, 1982, pp. 6-13.

[BOURICIUS 69] Bouricius, W. G., W. C. Carter, and P. R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," *Proceedings of the 24th ACM Annual Conference*, 1969, pp. 295-309.

[BREUER 76] Breuer, M. A. and A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Inc., Potomac, Maryland, 1976.

[CHEN 78] Chen, L. and A. Avizienis, "N-Version Programming: A Fault Tolerant Approach to Reliability of Software Operation," *Proceedings of the International Symposium on Fault Tolerant Computing*, 1978, pp. 3-9.

[CHOI 92] Choi, C. Y., "WPc: A Software Package for the Dependability Analysis of Powered Wheelchair Systems," Master of Science Thesis, University of Virginia, Department of Electrical Engineering, Charlottesville, Virginia, May 1992.

[CLARK 92] Clark, K. A. and Johnson, B. W., "A Fault-Tolerant Solid-State Memory for Spaceborne Applications," *Proceedings of the Government Microelectronics Applications Conference (GOMAC)*, November 9-12, 1992, Las Vegas, Nevada.

[FORTES 84] Fortes, J. A. B. and C. S. Raghavendra, "Dynamically Reconfigurable Fault-Tolerant Processing Arrays," *Proceedings of the 14th Annual International Symposium on Fault-Tolerant Computing*, Kissimmee, Florida, June 20-22, 1984, pp. 386-392.

[HAMMING 50] Hamming, R. W., "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, Vol. 26, No. 2, April 1950, pp. 147-160.

[HAYES 85] Hayes, J. P., "Fault Modeling," *IEEE Design and Test*, Vol. 2, No. 2, April 1985, pp. 88-95.

[HERBERT 83] Herbert, E., "Computers: Minis and Mainframes," *IEEE Spectrum*, Vol. 20, No. 1, January 1983, pp. 28-33.

[HOPKINS 78] Hopkins, Jr., A. L., T. B. Smith, III, and J. H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1221-1239.

[JOHNSON 84] Johnson, B. W. and P. M. Julich, "Reliability Analysis of the A129 Integrated Multiplex System," *Proceedings of the National Aerospace and Electronics Conference (NAECON)*, Dayton, Ohio, May 1984, pp. 1229-1236.

[JOHNSON 85] Johnson, B. W. and P. M. Julich, "Fault Tolerant Computer System for the A129 Helicopter," *IEEE Transactions on Aerospace and Electronic Systems*, Vol. AES-21, No. 2, March 1985, pp. 220-229.

[JOHNSON 87] Johnson, B. W. "A Course on the Design of Reliable Digital Systems," *IEEE Transactions on Education*, Vol. E-30, No. 1, February 1987, pp. 27-36.

[JOHNSON 89] Johnson, B. W., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[KATZMAN 77] Katzman, J. A., "System Architecture for NonStop Computing," *Proceedings of the 14th Computer Society International Conference (Compcon)*, San Francisco, February 1977, pp. 77-80.

[KOHAVI 78] Kohavi, Z., *Switching and Finite Automata Theory*, McGraw-Hill Book Company, New York, New York, 1978.

[LALA 85] Lala, P. K., *Fault Tolerant and Fault Testable Hardware Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[LAPRIE 85] Laprie, J-C., "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Proceedings of the 15th Annual International Symposium on Fault-Tolerant Computing*, Ann Arbor, Michigan, June 19-21, 1985, pp. 2-11.

[LIN 83] Lin, S. and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[NELSON 82] Nelson, V. P. and B. D. Carroll, "Fault-Tolerant Computing (A Tutorial)," presented at the AIAA Fault Tolerant Computing Workshop, November 8-10, 1982, Fort Worth, Texas.

[NELSON 86] Nelson, V. P. and B. D. Carroll, *Tutorial: Fault-Tolerant Computing*, IEEE Computer Society Press, Washington, D. C., 1986.

[PATEL 82] Patel, J. H. and L. Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Transactions on Computers*, Vol. C-31, No. 7, July 1982, pp. 589-595.

[RENNELS 80] Rennels, D. A., "Distributed Fault-Tolerant Computer Systems," *IEEE Computer*, Vol. 13, No. 3, March 1980, pp. 55-64.

[SHOOMAN 68] Shooman, M. L., *Probabilistic Reliability: An Engineering Approach*, McGraw-Hill Publishing Company, 1968.

[SIEWIOREK 82] Siewiorek, D. P. and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, Massachusetts, 1982.

[SKLAROFF 76] Sklaroff, J. R., "Redundancy Management Technique for the Space Shuttle Computers," *IBM Journal of Research and Development*, Vol. 20, No. 1, January 1976, pp. 20-28.

[STIFFLER 76] Stiffler, J. J., "Architectural Design for Near-100% Fault Coverage," *Proceedings of the International Symposium on Fault Tolerant Computing*, Pittsburgh, Pennsylvania, June 21-23, 1976, pp. 134-137.

[TANG 69] Tang, D. T. and R. T. Chien, "Coding for Error Control," *IBM Systems Journal*, Vol. 8, No. 1, January 1969, pp. 48-86.

[TOY 78] Toy, W. N., "Fault-Tolerant Design of Local ESS Processor," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1126-1145.

[TRIVEDI 82] Trivedi, K. S., *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[WADSACK 78] Wadsack, R. L., "Fault Modeling and Logic Simulation of CMOS and MOS Integrated Circuits," *The Bell System Technical Journal*, Vol. 57, No. 5, May-June 1978, pp. 1449-1475.

[WENSLEY 78] Wensley, J. H., et. al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol. 66, No. 10, October 1978, pp. 1240-1255.

## 1.6. Problems

**1.1.** Explain the differences between faults, errors, and failures, and explain how each term relates to the three-universe model. Illustrate your explanation with an example.

**1.2.** Some systems are designed for high reliability while others are designed for high safety. Explain the difference between reliability and safety and give an example of an application requiring high reliability and another example which requires high safety. How would you expect the four causes of faults to complicate the designs of high reliability applications as compared to high safety applications?

**1.3.** Faults can be characterized by five major attributes. Give examples of faults that illustrate each of these attributes.

**1.4.** Fault masking is an attractive technique for use in systems that cannot allow even momentary erroneous results to be generated. However, fault masking does have several serious limitations. What are the disadvantages of using a fault masking approach.

**1.5.** An industrial controller is needed to maintain the temperature of a process during a chemical reaction. The non-redundant controller is fairly simple and consists of an analog-to-digital (A/D) converter, processor, and a digital-to-analog (D/A) converter. Develop two alternatives for making the controller tolerant of any two component failures. The term component here means an A/D, processor, or D/A. Show block diagrams of your approaches and compare them. Which approach would you recommend and why?

**1.6.** Show the organization of an eight-bit memory with Hamming single error correction and double error detection. Be explicit in your descriptions and show the parity groups that result. Associate the syndromes with the particular bit that they identify as erroneous.

**1.7.** Show the separable and nonseparable cyclic code words that result for four-bit information words when the generator polynomial is $G(X)=1+X+X^2+X^5$. Also, develop a circuit that is capable of encoding the original information and a second circuit that is capable of decoding the code words.

**1.8.** Calculate the *MTTF* of a *TMR* system that contains three identical modules, each with a failure rate of $\lambda$ failures hour. You may assume that the modules obey the exponential failure law. Compare the *MTTF* of the *TMR* system with the *MTTF* of a single module having the same failure rate. Show a plot of each *MTTF* versus $\lambda$.

**1.9.** Construct the Markov model of a *TMR* system with a single spare. Incorporate a coverage factor associated with the process of identifying the failed module and switching in the spare. Assume that the spare is always powered and is just as likely to fail as the primary modules. If the failure rate of each module is 0.001 failures per hour, what is the reliability of the *TMR* system with the single spare at the end of a ten hour time period, as a function of the fault coverage factor? If the coverage is perfect, how does the reliability of the *TMR* system with the single spare compare with the reliability of a *TMR* system without a spare (again at the ten hour time period)?

**1.10.** The architecture of a flight control system is simply a *TMR* system that uses flux-summing as the voting mechanism. Each of the three processors in the system performs self-diagnostics to allow faults to be detected. If a fault is detected, the affected processor will remove itself from the flux-summing arrangement. Because of concerns about the safety of the system it has been decided to perform a safety analysis of the proposed architecture. Construct a Markov model of the system and develop a safety analysis. Compare the reliability and safety of the system as functions of the fault coverage factor associated with the self-diagnostics.

**1.11.** A satellite memory system uses a technique known as column sparing to improve the reliability. Specifically, each *k*-bit memory word is expanded by *s* spare memory cells. So, each of the n memory rows contains *k*+*s* memory cells. Assume that you have *n* by 1 (*n* rows with each row having one memory cell) memory chips that have a failure rate of $\lambda$. Also, assume that each memory chip obeys the exponential failure law. Write an expression for the reliability of the complete memory system which uses column sparing. The expression must account for fault coverage. Suppose that *k*=16, fault coverage is 0.95, and $\lambda = 0.915\times10^{-5}$ failures per hour. Determine the number of spares, *s*, which will maximize the reliability of

the memory at the end of a ten-year space mission. What is the reliability of the memory at the end of ten years (for both zero spares and the optimal number of spares)?

**1.12.** A possible alternative to the traditional hybrid *TMR* system with a spare (three operational units and one spare to replace any one of the three) is to use *TMR* and provide a spare for one, and only one, of the modules. In other words, the spare can only replace module 3, for example. Using combinatorial modeling techniques develop an expression for the reliability of such a system. Assume that the voter circuitry and the reconfiguration circuitry, including coverage, are perfect. Also assume that each module has a failure rate of $\lambda$ and obeys the exponential failure law. Using a failure rate of 0.001 failures per hour, compare the reliability of this hybrid system to traditional *TMR* with no spares (compare the reliabilities at three or four time points).

**1.13.** In many digital systems transient failures are much more prevalent than permanent ones, particularly when the operating environment for the system is extremely harsh. Consequently, it is very important to be able to analyze the effect of transients. Assume that you have an electronic device (say, a microprocessor) that has a permanent failure rate of $\lambda_p$ and a transient failure rate of $\lambda_t$. Further assume that the device has built-in detection capabilities that provide a coverage factor of $C_p$ for permanent failures and $C_t$ for transients. Finally, assume that the repair rate for transient failures is $\mu_t$, and the repair rate for permanent failures is $\mu_p$. Develop a discrete-time, discrete-state Markov model that accounts for both permanent and transient failures and allows the availability of the device to be calculated. Show the state diagram of the model, and write the state equations (in matrix form) that describe the model. You do not have to solve the model.

**1.14.** A satellite memory system uses an Error Detecting And Correcting (*EDAC*) code as a means of overcoming bit errors due to alpha particles. Essentially, alpha particles can occur at random times and cause random bits to change state in the memory. The memory cells are not permanently damaged by the alpha particles, but the state of the cell is changed. The memory uses a Hamming code which can correct all single errors and detect all double errors. The memory contains $n$ words with each word having $k$ data bits and $c$ Hamming check bits. Assume that the memory cells have a "hit rate" of $h$ bit errors per day-cell. The hit rate is the rate at which alpha particles are expected to corrupt the memory cells, assuming that the hits occur according to a Poisson process. You may assume that all bit errors are independent. Determine an expression for the probability that the memory with the *EDAC* will produce an uncorrectable bit error in any one of its $n$ words. Also, write the expression for the memory without the *EDAC* coding. Suppose that $n$ = 134,217,728 (128 megawords), $k$ = 16, $c$ = 6, and $h$ = 0.0000027 bit errors per day-cell. What is the probability of an uncorrectable bit error occurring after ten years of operation in space for both the memory with the *EDAC* and the memory without the *EDAC*. Remember that if the *EDAC* is not used, each memory word will have exactly $k$ cells.

**1.15.** One of the most difficult problems with duplication with comparison is the inability to detect faults that occur in the comparator. Using your knowledge of the four types of redundancy (and using any type redundancy you want) develop a gate-level design for a two-bit comparator (it compares two two-bit binary numbers) with two outputs. The combination of the two outputs should identify whether or not the two input numbers agree and whether or not the

comparator has a fault. For example, you might use one output to report on the comparison and the other to report the health of the comparator. Or, you might choose to encode the outputs in such a way that 00 (just as an example) implies the two numbers agree and the comparator is fault-free. Any stuck-at-1 or stuck-at-0 fault should be detected if it, in any way, affects the output of the comparator. Use your "self-checking comparator" to design a full adder that uses duplication with comparison.

**1.16.** Suppose that a bus between a microprocessor and a memory is 32-bits wide and is protected by four interlaced parity bits. Parity bits 1 and 3 are odd parity, and parity bits 3 and 4 are even parity. Show the organization of the parity groups and the design of the parity generation circuits. Show specifically which bits are in which parity groups. Describe the types of errors that are detected by this particular scheme, and give an example of each type. For example, does the approach detect all single-bit errors? What types, if any, of double-bit errors does the approach detect? Does the approach detect the all-ones and all-zeros cases?

**1.17.** Standby sparing is a very popular form of redundancy used in many fault-tolerant designs. Consider a simple system containing two processors that operate in a standby sparing arrangement. One processor is designated as the on-line processor and performs all of the system's computations, as long as it remains fault-free. The second processor serves as a backup that is only used in the event of a fault in the on-line processor. Both the on-line and the spare processors execute self-test routines which attempt to detect faults that occur. If, and only if, a fault is detected in the on-line processor, the spare processor is brought on-line to assume the functions of the system. If faults are detected in both processors, the system shuts down. The purpose of this problem is to compare the effectiveness of the standby sparing approach with a nonredundant (simplex) system. For the simplex system you may assume that it also has self diagnostics, but it does not possess the attribute of fault tolerance (since there is no redundant processor). If a fault is detected in the simplex system, it will simply shut down. How does the reliability of these two approaches vary as a function of fault coverage (this question is best answered by developing a combinatorial model of each system and sketching the reliability function versus coverage)? Suppose that we define safety as the probability that the system is either functioning correctly or has performed a successful shutdown. What value does the safety approach as time approaches infinity for each of the two architectures? You may assume that the coverage provided by the self diagnostics, in both the simplex and the standby sparing cases, is some arbitrary number *C*. Also, you may assume that the processors have a constant failure rate ($\lambda$) and obey the exponential failure law.

**1.18.** Suppose that a simplex (no redundancy) computer system has a failure rate of $\lambda$ and a fault detection coverage of *C*. The fault detection capability is the result of self-diagnostics that are run continuously. If the self-diagnostics detect a fault, the time required to repair the computer system is 24 hours because the faulty board is identified, obtained overnight, and easily replaced. If, however, the self-diagnostics do not detect the fault, the time required to repair the system is 72 hours because a repair person must visit the site, determine the problem, and perform the repair. The disadvantage of including the self-diagnostics, however, is that the failure rate of the computer system becomes $\alpha\lambda$. In other words, the failure rate is increased by a factor of $\alpha$ because of the addition of the self-diagnostics. Using analytical techniques

and a Markov model, determine the value of $\alpha$, for a coverage factor of 0.95, at which including the self-diagnostics begins to degrade the steady-state availability of the system.