A Network Element Based Fault Tolerant Processor

by

Todd A. Abler

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science in
Electrical Engineering and Computer Science

and

Bachelor of Science in
Electrical Science and Engineering

at the

Massachusetts Institute of Technology

May 1988

©Todd A. Abler, 1988

The author hereby grants to M.I.T. permission to reproduce
and to distribute copies of this thesis in whole or in part.

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
May 6, 1988

Certified by ______________________________
William J. Dally
Thesis Supervisor

Certified by ______________________________
Richard E. Harper
Charles Stark Draper Laboratory

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUL 26 1988

LIBRARIES

ARCHIVES

A Network Element Based Fault Tolerant Processor

by

Todd A. Abler

Submitted to the Department of Electrical Engineering and Computer Science on May 6, 1988 in partial fulfillment of the requirements for the degrees of
Master of Science
and
Bachelor of Science

Abstract

Fault tolerant computer architectures have typically increased reliability at the cost of performance. An architecture is proposed herein which satisfies the requirements for Byzantine failure resilience in an efficient manner, such that high reliability can be obtained at a higher performance level than that previously achieved. The architecture is based on a network element (NE) which provides a solution to the synchronization, communication and redundancy management requirements for malicious failure resilience. The NE is relatively compact and efficient, substantially reducing the performance cost of implementing fault tolerance.

Thesis Supervisor: William J. Dally
Title: Professor of Electrical Engineering

Acknowledgments

This work was made possible by the expertise and support of the staff of the Systems Architecture Division of the Charles Stark Draper Laboratory. In particular, special thanks must be extended to Jay Lala, Richard Harper, Stuart Adams, and Steven Friend who have all contributed significantly to the development of this thesis. Additionally, I would like to thank Ross Dettmer for his aid in implementing and evaluating the NEFTP. Finally, I thank my thesis advisor William Dally for his assistance and direction.

This work was done at the Charles Stark Draper Laboratory under an internal research and development grant.

Publication of this report does not constitute approval by the Draper Laboratory of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to the Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

Todd A. Abler

Charles Stark Draper Laboratory hereby grants permission to the Massachusetts Institute of Technology to reproduce and to distribute this thesis in whole or in part.

To Newtowne Variety - an oasis of excellence.

Contents

1 Introduction **10**
- 1.1 Problem Statement 10
- 1.2 Objective 10
- 1.3 Approach 10

2 Fault Tolerance Fundamentals **12**
- 2.1 Quantification of a System's Fault Tolerance 13
- 2.2 Motivation for Fault Tolerant Computing 13
- 2.3 Two Fundamental Approaches to Fault Tolerance 14

3 Byzantine Resilience **16**
- 3.1 Rational of the Byzantine Resilience Requirements 17
  - 3.1.1 Synchronization 17
  - 3.1.2 Connectivity 17
  - 3.1.3 Participants and Communication Rounds 18
- 3.2 Byzantine Resilient Architectures 21
  - 3.2.1 The Software Implemented Fault Tolerance Computer (SIFT) 24
  - 3.2.2 The AIPS Architecture Fault Tolerant Processor (AIPS FTP) 25
  - 3.2.3 The Multicomputer Architecture for Fault Tolerance (MAFT) 28
- 3.3 The Network Element Based Fault Tolerant Processor (NEFTP) . . 29

4 NEFTP: The Network Element Design **34**
- 4.1 The Basic NE cycle 34
- 4.2 The NE Functional Sub-Sections 35
  - 4.2.1 The PE/NE Interface 36
  - 4.2.2 The NE Data Paths 41
  - 4.2.3 Inter-FCR Communication Links 48
  - 4.2.4 The NE Fault Tolerant Clock 49
  - 4.2.5 The NE Scoreboard 56
  - 4.2.6 The NE Global Controller 60
- 4.3 Status of The NE Prototype 66

5 Evaluation of The NEFTP **68**
- 5.1 Reliability Analysis 68
  - 5.1.1 Definitions and Assumptions 68
  - 5.1.2 Simplex System Reliability Analysis 70
  - 5.1.3 NEFTP Reliability Analysis 73
- 5.2 Performance Evaluation 81

5.2.1 NEFTP Analysis 83
5.3 NEFTP Evaluation Summary 89

6 Conclusions and Recommendations **90**

A NE Schematics **94**
A.1 Sheet 1 94
A.2 Sheet 2 95
A.3 Sheet 3 96
A.4 Sheet 4 97
A.5 Sheet 5 98
A.6 Sheet 6 99
A.7 Sheet 7 100
A.8 Sheet 8 101
A.9 Sheet 9 102
A.10 Sheet 10 103
A.11 Sheet 11 104
A.12 Sheet 12 105
A.13 Sheet 13 106

B PLD Equations **107**
B.1 VME Interface 107
B.1.1 Address Decoding 107
B.1.2 VME Block Transfer Address Counter 107
B.1.3 Data Strobe Generator 108
B.1.4 DTACK Generator 109
B.1.5 VME FIFO Control 1 111
B.1.6 VME FIFO Control 2 112
B.2 Fault Tolerant Clock 112
B.2.1 MYFTC Generator 112
B.2.2 Median Bound Voter and Clock Error Accumulator 117
B.2.3 Bound Generator 120
B.2.4 System Clock Generator 121
B.3 Data Paths 122
B.3.1 Data Path Voter Slice 122
B.3.2 Debug Router 124
B.3.3 Synchronous Data Path Controller 125
B.3.4 Asynchronous Data Path Controller 126
B.3.5 Syndrome Accumulator 128
B.3.6 Link Error Accumulator 129
B.4 Scoreboard 131
B.4.1 Scoreboard First Half 131
B.4.2 Scoreboard Second Half 134

B.5 Global Controller 136
B.5.1 Multiplexor First Half 136
B.5.2 Multiplexor Second Half 137
B.5.3 Next State Register 138
B.5.4 Controller Decoder 139
B.5.5 Mask, Size, and Debug Register 140
B.5.6 Data Path Voter Mask Register 143
B.5.7 Global Controller Event Counter 145

List of Figures

2.1 Performability of Selected Architectures 13

3.1 FCRs Connected via Shared Link 18
3.2 Self Checking Pair Architecture 19
3.3 TMR Architecture 19
3.4 TMR Architecture with Simplex Sensor 20
3.5 Canonical 1-Byzantine Resilient Architecture 22
3.6 AIPS Fault Tolerant Processor (Triplex) 26
3.7 MAFT Architecture 28
3.8 NEFTP Architecture 29

4.1 BRVC Abstraction 34
4.2 Basic NE Cycle 35
4.3 PE/NE Interface Block Diagram 36
4.4 NE Exchange Types 38
4.5 NE Memory Map 40
4.6 NE Data Paths Block Diagram 42
4.7 NE Physical and Virtual Identifiers 43
4.8 Inter-FCR Communication Link Block Diagram 48
4.9 Basic Message Transmission Frame Signals of FTC 49
4.10 FTC Block Diagram 50
4.11 Self Normal FTC Timing Signals of FTC 52
4.12 Self Ahead FTC Timing Signals of FTC 52
4.13 Self Behind FTC Timing Signals of FTC 53
4.14 State Diagram of MYFTC Generator 54
4.15 Basic Message Transmission Frame; External Signals 55
4.16 SERP Packet Format 56
4.17 Scoreboard Block Diagram 57
4.18 NE Operation Time Line 59
4.19 SERP Packet Delivered to Scoreboard: SERP 2 59
4.20 SERP Packet Delivered to Scoreboard: SERP 3 60
4.21 NE Global Controller Block Diagram 61
4.22 Mask, Size and Debug Register Latch Functions 66
4.23 Global Controller Control Outputs 67

5.1 Simplex Processor Markov Model 71
5.2 p_{unsafe}, $p_{shutdown}$, and $p_{unsafe} + p_{shutdown}$ For A Simplex Processor . . 71
5.3 p_{unsafe} vs c For A Simplex Processor 72
5.4 $p_{sysloss}$ vs f_t For A Simplex Processor 72

5.5 NEFTP Markov Model: Redundancy Scheme I 74
5.6 $p_{unsafe}, p_{shutdown}$, and $p_{unsafe} + p_{shutdown}$ For NEFTP Redundancy Scheme I 75
5.7 Addition To NEFTP Markov Model For Redundancy Scheme II . . . 76
5.8 $p_{unsafe}, p_{shutdown}$, and $p_{unsafe}+p_{shutdown}$ For The NEFTP Redundancy Scheme II 77
5.9 Addition to NEFTP Markov Model For Redundancy Scheme III . . 79
5.10 $p_{unsafe}, p_{shutdown}$, and $p_{unsafe}+p_{shutdown}$ For The NEFTP Redundancy Scheme III 80
5.11 Reliability Analysis Summary 81
5.12 Typical Control Loop 82
5.13 NE Data Exchange Times and Throughput 84
5.14 Processor Pre-Synchronization Skew as a Function of Synchronization Frequency 85
5.15 Fault Tolerance Related Data Exchange Overhead as a Percentage of Control Loop Length 86
5.16 Breakdown of Fault Tolerance Related Data Exchange Overhead When Using Maximum Packet Size 87

Chapter 1
Introduction

1.1 Problem Statement

A requirement for ultra-reliable computation has generated much research in the area of fault tolerant computing. Several fault tolerant computer architectures have been designed, implemented, and evaluated. These include the following:

1. Software Implemented Fault Tolerance (SIFT) computer [Wen78].
2. Multi-Computer Architecture for Fault Tolerance (MAFT) [WKF85] [KWFT88].
3. Advanced Information Processing system Fault Tolerant Processor (AIPS FTP), and its derivatives [Lal84,Lal86,LAGD86,Smi83].

All such architectures have increased reliablility through use of component redundancy. Typically this increased reliability has been accompanied by rigid programming constraints, substantial computational overhead, and/or significant additional hardware.

As will be shown later, many of the constraints imposed by the above architectures are not inherent in the theoretical requirements for failure resilience. The problem, therefore, is to design a fault tolerant processor architecture with a minimum loss of performance and flexibility.

1.2 Objective

The primary objective of this thesis is to design and develop a fault tolerant processor which does not unnecessarily constrain the programming model, and which has minimal computational and hardware overhead. Specifically, this involves understanding the necessary and sufficient requirements for failure resilience. An architecture is developed which incorporates these requirements. An implementation of this architecture is designed and fabricated. A secondary objective is to quantitatively illustrate the benefits of this architecture via performance and reliability comparisons with prior fault tolerant processor designs.

1.3 Approach

This study begins with a discussion of the basics of fault tolerant computing. Chapter 2 presents some of the background and terminology of fault tolerance.

Some examples motivating the necessity of fault tolerant computing are given. The current state of research in fault tolerant computing demonstrates the existence of two different fundamental approaches. Chapter 2 discusses the two approaches and examines their differences. Only one of the approaches, known as Byzantine failure resilience, is shown to be a viable solution to the problem. Byzantine resilience and its theoretical requirements are discussed in Chapter 3. It is shown how existing architectures, such as those mentioned above have satisfied these requirements; yet in doing so have placed severe, and perhaps unnecessary, constraints on the problem. The Network Element (NE) is introduced as a facility which provides for the requirements of Byzantine failure resilience in an efficient manner. The anticipated advantages of this architecture are suggested. Chapter 4 gives a detailed discription of the Network Element design that has been incorporated into a Network Element based Fault Tolerant Processor (NEFTP). The performance and reliability of the NEFTP is investigated in Chapter 5. This thesis concludes with some discussion of the actual performance and reliability measures of the NEFTP.

Chapter 2
Fault Tolerance Fundamentals

This chapter is presented as an introductory overview of fault tolerance, and as such is not rigorously complete. It is hoped that the information presented herein will be sufficient for the purposes of this thesis. For situations in which this is not the case, more information may be found in the references.

Fault tolerant computing is defined as "the ability to execute specified algorithms regardless of hardware failures, total system flaws or program fallacies [Kim75]." A collective term for "hardware failures, total system flaws or program fallacies" is "faults." As implied by the definition, faults can be of a hardware or software nature. Hardware faults can either be permanent, intermittent or transient [SS82]. A fault may be active, currently manifesting erroneous behaviour in a part of the system, or a fault may be latent, producing no erroneous behavior at the given momment but having the potential to do so in the future.

Fault tolerance is achieved through the use of redundancy such that faults can be detected and corrected prior to the entrance of the aggregate logical machine into an erroneous state, or propagation of errors to the system outputs. This redundancy can be of many forms. Informational redundancy, such as over-specified simultaneous equations and error-correcting codes, can be used to detect and correct faults in a limited manner [Fri86]. However, informational redundancy schemes are difficult to devise to detect and correct ALU faults [SS82]. Hence, informational redundancy is ineffective against total system flaws since it cannot provide total system coverage. Temporal redundancy provides a means of tolerating transient hardware faults by re-executing an instruction stream several times [LS88]. Clearly temporal redundancy cannot tolerate permanent hardware failures. Hardware redundancy allows tolerance of permanent and transient hardware faults which are uncorrelated between the redundant components (correlated hardware faults can affect redundant elements simultaneously, and hence cannot be tolerated if redundant elements are identical). Software redundancy may be employed to tolerate uncorrelated software errors. Design deversity on behalf of redundant components is necessary (though not necessarily sufficient [KLJ85,AK84]) to protect against correlated faults.

Fault tolerant computers can be differentiated in terms of the type of faults tolerated, the definition and level of tolerance, and the architecture (type of redundancy) employed to provide the tolerance. In order to compare the types of fault tolerant computers there must exist some means of quantifying the fault tolerance of a system.

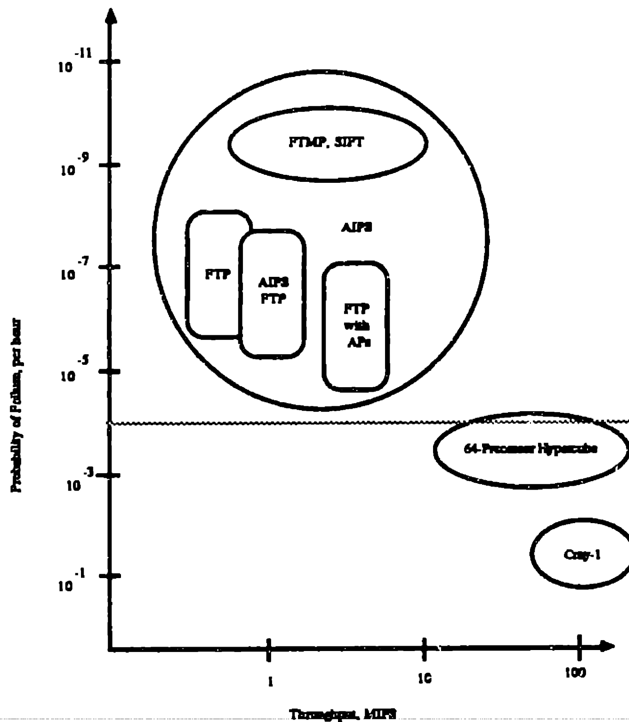


Figure 2.1: Performability of Selected Architectures

2.1 Quantification of a System's Fault Tolerance

The fault tolerance of a system can be measured in terms of its *reliability*, *availability*, and *coverage*. The reliability of a system is expressed in terms of the probability that it functions correctly for a given duration. The less strict metric availability is the probability that the system will be able to function correctly at a given time (roughly corresponding to the ratio of the system failure rate to system repair time). Coverage is expressed in terms of the probability that the system will function correctly given the occurrence of an arbitrary fault. Recent study has suggested an interesting new metric called *performability* which is useful as a figure of merit of fault tolerant systems [Har87]. One way to express performability involves mapping systems onto a cartesian coordinate system which has performance (throughput) as the abscissa, and reliability as the ordinate. Figure 2.1 shows the performability of selected systems [Har87].

2.2 Motivation for Fault Tolerant Computing

There exist many applications today that require significant computation critical to the success of the task. A fault tolerant system must provide reliability at a level commensurate with the computation's criticality. Less stringent applications, such as electronic banking and investment, may allow for operator intervention to repair

or replace failed components with a finite interruption in service. Some applications, such as telephone swithching services, allow for operator intervention with no interruption in service. An area illustrating stringent requirements is in autonomous and semi-autonomous vehicle control. Here no operator intervention may be possible and the results of a system failure may be catastrophic. The reliability requirements of such a system are typically set at a failure rate 10^{-7} per hour for military applications and 10^{-10} failures per hour for civilian applications, [Bro87,Wen78] . A non-fault tolerant system composed of hundreds of typical VLSI components each having a failure rate of 10^{-7} permanent failures per hour [Gol84], and other failure sources, is incapable of satisfying this reliability requirement. Therefore fault tolerance must be incorporated into computer architectures which perform these applications.

2.3 Two Fundamental Approaches to Fault Tolerance

Neither informational redundancy nor temporal rendundancy can satisfy the reliability requirements mentioned above; this is a direct consequence of the fact that these redundancy schemes are incapable of tolerating failures of any system components with unity coverage. Hence, current efforts in ultra-reliable computing employ hardware redundancy. There currently exist two fundamentally different approaches to providing hardware redundancy for fault tolerance.

The approach taken by much of private industry is to use minimal hardware redundancy in designing systems which provide tolerance of only those failure modes which are predefined as "likely". This approach is *ad hoc* in nature and thus will be refered to as the ad hoc approach. The ad hoc approach is not based on any theory. It involves performing some failure mode analysis of the target system to determine the likelihood of *a priori* determined failure modes. Hardware redundancy can then be designed in to provide coverage of those failure modes with a sufficiently high probability of occurrence. System reliability is then related to the sum of the probabilities of the uncovered failure modes. For example, suppose a failure modes analysis of a given system yielded only three likely failure modes a, b, and c with probabilities of occurrence per unit time f_a, f_b, and f_c respectively. If unity coverage was provided for these modes, then, purportedly, the probability of system loss per unit time would be no more then $1 - (f_a + f_b + f_c)$.

The approach taken by the academic and defense research communities is that sufficient hardware redundancy must be employed to provide tolerance of all failure modes. Tolerance of arbitrary failure modes is known as Byzantine resilience. This is the design philosophy of the NEFTP and of those architectures to which it is compared. A detailed discussion of Byzantine resilience will be given in Chapter 3.

The validity of the ad hoc approach is seriously questioned when considering the assumptions, human biases and impracticalities involved. There are serious problems to validating or certifying such an architecture. To accurately assess the

reliability of such a system, lifetime testing is not possible. In other words, to know if the system has a probability of failure of less than 10^{-9} per hour, it is not feasible to build 100 identical systems and test them for 10^9 hours. Therefore, the only way to arrive at accurate reliability values using this approach would be to enumerate all possible failure modes. Yet digital systems have infinitely many failure modes. Pattern sensitive random access memory (RAM) failures evidence this fact. In fact, the only way to certify and validate a fault tolerant architecture is via analytical modeling coupled with empirical testing. Yet the ad hoc approach is not based on any theory and hence is not amenable to analytical modeling. These are serious problems. In addition to the above problems, most architectures produced by the ad hoc approach provide no coverage for malicious failures. Malicious failures just do not happen (almost). It is extremely foolhardy to dismiss malicious failures by saying that they just do not happen. This is incorrect. In [MG78], an in-flight failure of a non-Byzantine resilient architecture was recorded and attributed to lack of safeguards against malicious failure. Though one cannot measure how likely malicious failures are, one can estimate how unlikely they must be in order that a non-Byzantine fault tolerant computer would successfully complete its application. Such a calculation for a future space application illustrates that malicious processor failures would have to have a likelihood of less than 10^{-6} that of other processor failures [Joh88] in order that sufficient reliability could be achieved in an architecture with no safeguards against them.

For these reasons Byzantine resilience can be seen to be the only approach to fault tolerance capable of meeting the reliability needs of life or mission critical applications. In order to provide tolerance to abitrary failure modes, a system must satisfy some stringent theoretical constraints (illustrated in the next chapter). This has typically resulted in architectures which have significantly increased cost and decreased performance compared to their non fault tolerant peers. The NEFTP architecture is anticipated to offer the high reliability of Byzantine resilience with a lower cost and performance penalty.

Chapter 3
Byzantine Resilience

The notion of Byzantine resilience derives from the solution of a distributed consensus protocol problem known as the Byzantine Generals Problem [LL82]. The problem scenario is as follows: several generals of the Byzantine army (one or more of whom may be traitors) are camped around an enemy city and can communicate only via messenger; derive a protocol to ensure that all non-traitorous generals agree on a common battle plan. This problem has been well studied, and theoretically demonstrable prerequisites for a solution exist. For a given number of traitor generals, the solution is constrained in the number of participating generals, the number of generals through which a message must be relayed, and to whom each general must relay the message. The prerequisites of a solution to the Byzantine generals problem are stated formally below.

A deterministic consensus protocol which is capable of correctly functioning in the presence of arbitrary behavior on behalf of f participants must meet the following requirements:

1. There must be at least $3f + 1$ participants [PSL80].

2. Each participant must be connected to at least $2f + 1$ other participants through unique communication links. [Dol82].

3. The protocol must consist of a minimum of $f+1$ communication rounds among participants [FL82].

4. The participants must be synchronized to within a known upper bound [DDS84].

An architecture which meets the above requirements is said to be *f-Byzantine Resilient*. Such an architecture is capable of achieving near unity failure coverage.

A participant in the above protocol is known as a *fault containment region* (FCR). A FCR is a region to which faults are contained to the extent that the probabilities that faults occur in different FCRs are statistically independent. Fault containment is provided by three architectural features:

- physical isolation
- electrical isolation, including independent power
- independent clocking.

Many of the designs that will be discussed in this thesis take great liberties with these architectual guides for sound fault containment region design. It is excusable to the extent that it is merely a convenience for implementing the proof-of-concept prototype. Occasions in which these guidelines are violated in prototype by design decisions that make it impossible to remedy the situation in an actual production version of the machine will merit some discussion.

3.1 Rational of the Byzantine Resilience Requirements

This section is included to lend some plausibility to the requirements of Byzantine failure resilience and is in no way intended as proof, formal or otherwise (the references listed by the requirements provide such proof). In doing so, the failure modes of some commercially popular non-Byzantine resilient architectures will be used as counter examples, and the canonical 1-Byzantine resilient architecture will be introduced.

3.1.1 Synchronization

Participating FCR's must be synchronized to within a known and bounded skew. No asynchronous protocol is capable of achieving consensus in the presence of failures because each such protocol has the possibility of non-termination [FLP85]. The rational behind the synchronization constraint is two-fold. First, some temporal bound must be placed on the behavior of a FCR in order that an inert failure in one FCR can be detected. Second, the mechanism used to compare values from participating FCRs must know when valid data has arrived from all non-faulty FCRs.

3.1.2 Connectivity

Unique communication links must be used to connect all participating FCRs. Without such links, a non-faulty FCR is not guaranteed of receiving valid information from all other non-faulty FCRs. This prohibits the participants from achieving consensus regardless of the number of communication rounds employed. For example, consider the situtation where two FCRs share a communication link to another FCR (Figure 3.1). It is possible for one of the transmitting FCRs (B) to fail in such a way as to corrupt messages from C to A. In this manner FCRs A and C effectively cannot communicate with one another; hence they will never be able to reach agreement on anything. Assuming for now that $3f + 1$ participants are required, then the constraint for unique communication links dictates that there be $2f + 1$ such links.

Protocols employing authenticated communication may be able to reduce the connnectivity constraint. Such authentication must ensure that a failure on behalf of a relaying node cannot corrupt message traffic through that node. Furthermore,

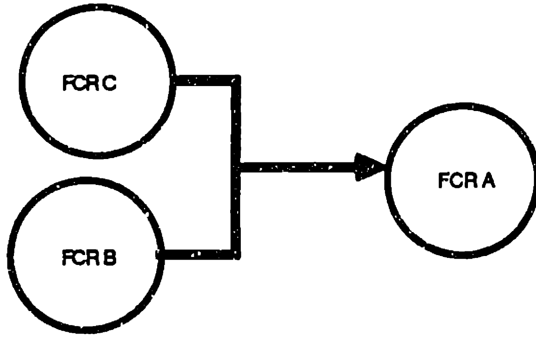


Figure 3.1: FCRs Connected via Shared Link

sufficient connectivity must still be provided to withstand the physical failure of f links.

3.1.3 Participants and Communication Rounds

The remaining two requirements can best be motivated by demonstrating the existence of failure modes in architectures which do not satisfy them. Two non-Byzantine resilient architectures will be used for this purpose: the self checking pair, and the triple modular redundancy (TMR) technique. Specifically, these architectures will be shown to be incapable of achieving consensus in the presence of arbitrary failures, whereas Byzantine resilient architectures are guaranteed to be able to achieve consensus. Consensus of a singly sourced input is known as input data consistency. Input data consistency necessitates the properties of *agreement* and *validity*. *Agreement* means that all non-faulty FCRs will reach agreement on the input value. *Validity* means that if the FCR which sources the input data is non-faulty, then all non-faulty FCRs will *agree* on the value actual sent by the source. Consensus of commonly sourced output is known as output consensus.

A typical self checking pair architecture (Figure 3.2) consists of two processors with a comparator attached to their outputs for output data comparison. Such an architecture clearly does not meet the requirments for minimal ($f = 1$) Byzantine resilience. There exists only two FCRs (at most three, if the comparator resides in its own FCR), when four are required. The requirement for number of participating FCRs will henceforth be known as the *cardinality requirement.* Similarly, this architecture does not satisfy the connectivity requirement. The two channels of a duplex cannot satisfy the property of agreement and hence cannot satisfy validity as well. This is a result of the *Two Generals Problem* [Gra79] which demonstrates that two generals who are isolated and initially undecided cannot reach consensus on a battle plan via communication through unreliable messangers. It can be shown that in order that consensus be reached at a given point in time, the generals must have been in agreement initially. An additional problem of this duplex architecture is that it is not capable of consensus of outputs, hence it is not fault masking. In the

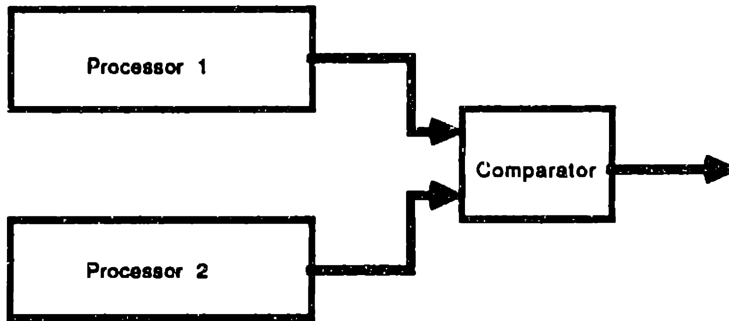


Figure 3.2: Self Checking Pair Architecture

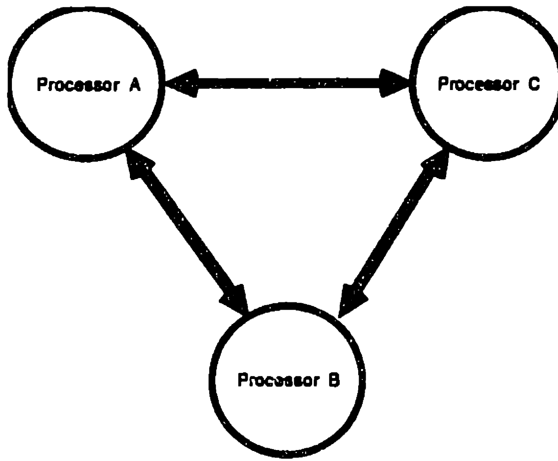


Figure 3.3: TMR Architecture

presence of a fault the comparator signals the discrepancy seen on its inputs; yet valid output is not assured because the comparator has no way of knowing which input is correct. Furthermore the comparator is a single point of failure; therefore, even if the comparator had some reasonable method of determining which FCR is correct, the comparator itself could fail and generate incorrect output. The reliability of the comparator is obviously an upper bound of the system reliability.

The architecture used in the TMR technique consists of three inter-connected processing sites (Figure 3.3). This architecture does not satisfy the cardinality and connectivity requirements of minimal Byzantine failure resilience. This renders it incapable of acheiving input data consistency. A TMR architecture with a simplex sensor (Figure 3.4) is used to illustrate the input consistency protocol in two scenarios.

Assume processor A hosts a simplex sensor, and the value of this sensor ($x; x \in (0, 1)$) needs to be distributed. In the first scenario processor A is faulty. An initial round of communication takes place and A sends a 0 to processor B and a 1 to processor C. Another round of communication is required in order that B tell C and A what A told it, and that C tell B and A what A told it. During this second

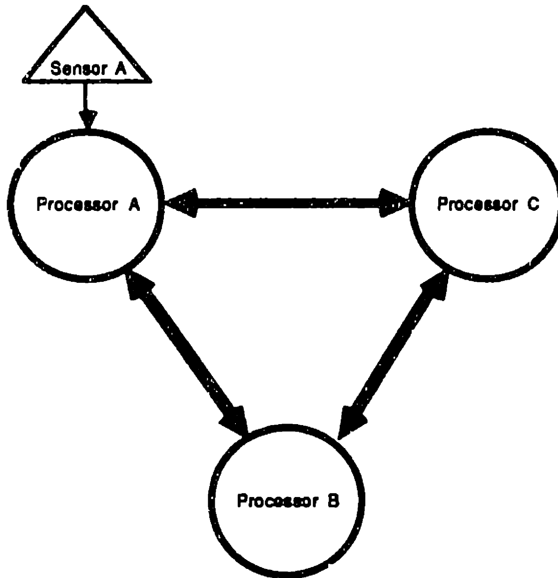


Figure 3.4: TMR Architecture with Simplex Sensor

round A does not source any data, because B and C have already heard what A had to say in the first round. At the end of the second communication round A, B and C have all the information that they can learn about the triad's status, hence no further communication rounds are required. At the end of the second communication round A, B and C both have a copy of the data set $(0, 1)$. Both B and C 'vote' this data set according to the following function

$$f(x_1, x_2) = x_1 \vee x_2$$

and arrive at the consistent result 1. The requirements for Byzantine failure resilience do not put any constraints on what happens in the faulty FCR A. The property of agreement of non-faulty FCRs is satisfied. Validity is not required since it is the source processor that is failed. This scenario evidences how an input consistency protocol would work for a TMR architecture. However, a second example illustrates the failings of this architecture.

The second scenario is the same as the first except that the failed processor is B, and that the actual sensor input is fixed as 0. The first exchange round takes place and A sends a 0 to B and C. The second round takes place and B sends C a 1 and A a 0, while C sends A and B both a 0. At the end of the second round the two non-faulty FCRs (A and C) have different data sets. A has the data set $(0, 0)$ while C has the data set $(0, 1)$. Applying these data sets to the voting function given above, A and C arrive at different results. Hence the properties of agreement and validity are violated. This is a disastrous problem. The non-faulty FCRs do not have a consistent view of the system status and subsequent computation in the two will therefore diverge. Furthermore the failed FCR B may actually recover the

valid input value causing the system as a whole to view FCR C as the source of the failure. An incorrect failure diagnosis has equally catastrophic ramifications for the system. This illustrates why the TMR architecture is not Byzantine resilient.

It should be pointed out that a cardinality of three would be sufficient for output consensus given input consistency. The output consensus protocol consists of a single communication round in which all FCRs broadcast their output value to all other FCRs. Thus in the presence of an arbitrary single fault all non-faulty FCRs will have a data set of three values of which at most one is incorrect. Hence a majority vote will yield agreement and validity of the output value among all non-faulty FCRs. Since output consensus is achieved in the presence of a single failure the TMR architecture is said to be a *Fault Masking Group* (FMG). The inherent problem here is that input consistency is not guaranteed. Therefore if an input (which is not guaranteed of being distributed consistently) is used to produce the output, then output consensus is not guaranteed.

The TMR example provides insight into the requirements of Byzantine resilience. The failure of the TMR architecture is in cardinality and connectivity. Two communication rounds are indeed sufficient for input data consistency for a minimal Byzantine resilient architecture. The problem is that at the end of the second round each FCR has only two values, one of which may be incorrect. Replicating sensors does not change this problem any. There will exist some finite skew between the sensor values from each FCR, and thus each must be exchanged through an input consistency protocol. The processors would each perform some mean value selection of their senor value set. As was shown in the example, no single sensor value can be distributed consistently, therefore the set of values is not guaranteed to be distributed consistently. The solution is to add another FCR such that each FCR will have three values at the end of the communication. Since at most one of the values can be incorrect, a majority vote function will guarantee agreement. Hence the 4 FCR requirement for minimal Byzantine resilience. The canonical 1-Byzantine resilient architecture is shown in Figure 3.5. This fourth FCR is not constrained to be a processing site. Only three processing sites are required to provide output consensus in the presence of a single fault. This fourth FCR can be a minimal hardware data exchanger capable of participating in the communication protocol.

3.2 Byzantine Resilient Architectures

Several architectures which satisfy the requirements for single Byzantine failure resilience have been designed and implemented. This section will analyze several of them in turn, illustrating how they meet the requirements of 1-Byzantine failure resilience. This discussion will form the foundation for the presentation of the NEFTP architecture. The NEFTP will be shown to have the property of Byzantine failure resilience, though it uses a synchronization scheme uncommon to other fault

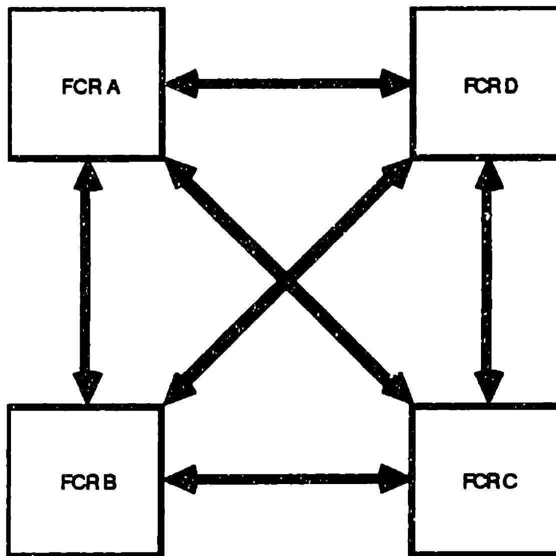


Figure 3.5: Canonical 1-Byzantine Resilient Architecture

tolerant processors.

Much of the discussion below will focus on the synchronization schemes employed by the various architectures. The synchronization mechanism of the NEFTP is seen to be the crucial ingredient in the reliability, flexibility, cost and performance benefits which are anticipated. Indeed this is true of all such architectures. The synchronization mechanism directly affects an architecture's performance, reliability and flexibility. To aid this discussion some issues of synchronization and the necessary terminology are discussed here.

Granularity of action is a parameter of synchronization schemes indicating the temporal difference between identical actions on behalf of different participants. Schemes which achieve *fine grain synchronization* have a small temporal difference among participants. Correspondingly, *coarse grain synchronization* schemes leave a large temporal difference between participants. Another parameter of synchronization schemes is *post-synchronization skew*. This refers to the minimum skew that the participants can attain after execution of the synchronization task. Note that post-synchronization skew is a fundamental lower bound on the skew between FCRs. Granularity of action is affected by *synchronization frequency* - the rate at which synchronizations take place. This is because skew builds up between FCRs, due to the drift of their respective clocks, during the periods between synchronizations. Given a synchronization mechanism, granularity of action can be reduced by synchronizing more frequently. This can be done until synchronizations are taking place continuously, at which point the maximum possible skew will be no less than the post-synchronization skew.

Why does granularity of action matter? As long as the participants are synchronized to within a known and bounded skew, consensus can be achieved. The

catch is that granularity of action directly impacts performance and reliability. The impact on performance is straightforward. The machine can output data no more frequently than the rate at which data can be exchanged and compared. The time required for data comparison cannot be shorter than the maximum possible skew between FCRs, as any FCR must wait until it is guaranteed to have data from all non-faulty FCRs in order to compare redundant copies. The impact of granularity of action on reliability is more subtle, but nonetheless important. In the short run the probability of system loss due to near simultaneous failures of redundant components is linearly proportional to the mean fault latency time plus the time it takes the system to reconfigure around the first fault (denote this sum as $1/\mu$). For a Byzantine resilient system, reconfiguring involves identifying and isolating a failed FCR by masking it out. This rate μ cannot be faster than the rate at which outputs from all participating FCRs can be compared. Outputs can be compared no faster than the rate of $1/maximumskew$. Hence the granularity of action of the synchronization mechanism directly affects the reliability; a finer granularity increases system reliability.

The choice of synchronization mechanism may also affect the architecture by imposing hardware and/or software constraints. These constraints are important not for implementation flexibility alone; they also impact reliability. An architecture which is free to employ design diversity in both the hardware and software of participating FCRs is arguably less susceptible to common mode design failures, hence more reliable. Design diversity in software may consists of scheduling tasks in different orders on different FCRs, or it may entail coding differently the versions of a task which are run on different FCRs (called *n-version programming*). Design diversity in hardware may consist of using single board computers from different vendors in different FCR's.

Some synchronization mechanisms impose a hardware constraint known as *clock determinacy*. This means that redundant processing sites must execute an identical number of instructions in a given number of clock cycles. To illustrate the importance of such a constraint, consider that clock determinacy constrains the hardware design of the processor tremendously, especially in light of the fact that commercially available computers are not clock deterministic. Typical problems include devices such as error correcting memories, floating point units, and memory managment units; where a given operation can take a variable number of clock cycles in the presence of a memory fault, overrun, or page fault. The system architect is left to design a clock deterministic processor by forcing worst case scenarios of such operations.

An example of a software constraint generated by the synchronization mechanism is when the applications programmer is required to know the maximum execution time of any task, and ensure that tasks are scheduled such that they are guaranteed to complete by a given time. The programming model of such a machine is quite more complex than a canonical simplex Von-Nueman machine. Correspond-

ingly the cost of code development is increased. However, this is a constraint that is typically imposed by the real time nature of such systems anyway.

3.2.1 The Software Implemented Fault Tolerance Computer (SIFT)

The SIFT architecture consisted of six processors configured with a point-to-point communication link between each pair of processors. This architecture obviously satisfies the cardinality and connectivity requirements. As will be explained below,the mechanism for synchronization and data comparision (i.e. voting) is provided by software. Aside from the somewhat specialized communication ports, the processors contain no hardware that is specific to fault tolerance.

The SIFT processors are synchronized by executing the *synchronization task*. Tasks must be allocated to *frames*, where each frame denotes a specified number of ticks of the local processors clock. Frames are approximately $100ms$ long, and are divided into 33 sub-frames of $3.2ms$ duration [PB86]. Each $1.6ms$ the processor is interrupted and decides whether to schedule a new task.

The synchronization task must be scheduled at the same hardware clock period on each processor. The synchronization task involves a processor broadcasting the value of its local clock to all processors during a set time window. Other windows denote a time for the processor to listen for the clock values broadcast from other participants. Upon receiving each value, the processor reads its local clock and notes the difference between the two. After all time windows have passed, a processor has a set of values representing the deviation of its local clock from all other processor clocks. Each processor uses these values to compute an adjustment to its local clock. A processor adjusts its local clock, and waits to begin the next frame when its local clock reads some *a priori* determined value.

Synchronization is achieved by this algorithm because fast processors will adjust their clocks such that they wait longer to begin the next frame than nominal processors. Correspondingly, slow processors will wait less time than nominal processors. No computation can be performed in the adjustment periods, as not all participants require the same number of them. This type of synchronization is known as *framewise synchronization* since adjustment is made at frame boundaries. The implementation of this algorithm has some significant drawbacks. Synchronization occurs with a low frequency (once a frame, or $100ms$) due in part to the fact that the synchronization task takes a long time to complete ($2ms$) [PB86]. Hence this scheme has a large maximum skew, though indeed bounded ($107\mu s$) [PB86]. The large possible maximum skew means that SIFT processors are *loosely synchronized*. The SIFT scheme is thus a coarse grain synchonization scheme. The method used by a SIFT processor to "read" another's clock value involves polling a memory location and waiting for the arrival of the broadcast clock value. This implementation involves some significant "read error" ($26\mu s$) [PB86]; the post-synchronization skew is directly related to this quantization error of the synchronization task. Therefore the post-synchronization skew is large.

The loose synchronization of SIFT causes both performance and reliability penalties. Perhaps a more important penalty is the programming constraint imposed by this scheme. The applications programmer must ensure that tasks are allocated to a frame such that the computation of a frame always takes an *a priori* determined number of sub-frames. This is done to ensure that the synchronization task can be scheduled at the identical sub-frame by each processor. The fault tolerance is not transparent to the applications programmer, making the cost of changing the application of the machine tremendous. This scheme does allow for limited hardware diversity since no constraints are placed on the hardware other than the (rather significant) constraint that it be possible to acheive bit-for-bit consensus of processor outputs. The scheme also allows for limited software diversity. Redundant sites are *framewise congruent* but they need not be *functionally congruent*. *Funtional congruency* means that the processors are not only synchronized as far as the number of tasks executed is concerned, but that the tasks executed also are functionally equivalent. It is possible in SIFT for redundant processors to schedule different functions in a given task slot, which arguably makes it less susceptible to transient errors.

SIFT implements data comparison with the *vote task*. A processor receives data from other processors in its memory-mapped mail box. The vote task then performs a n-way, $(4 < n < 6)$, bit-for-bit, maskable, majority vote on the redundant copies. The vote task also provides vote error information identifying any processor in disagreement. Such information is necessary to reconfigure around a permanently failed processor. This mechanism has some serious performance problems. A 5-way vote of a single data value (16 bit word) takes $413\mu s$ [PB86] in the absence of errors. This means that fewer than 8 values can be voted in a sub-frame.

As can be seen here, much of the throughput of a SIFT processor is dedicated to implementing the fault tolerance (1 sub-frame for synchronization, and many for data comparison, per frame). More will be said about this in Chapter 5.

3.2.2 The AIPS Architecture Fault Tolerant Processor (AIPS FTP)

The AIPS FTP takes an approach different than that of SIFT in two fundamental ways. First, a minimally Byzantine resilient configuration of the AIPS FTP consists of three processors and three additional FCRs called *interstages* (Figure 3.6). An interstage is a minimal hardware implementation of a Byzantine resilience mechanism. The motivation for interstages derives from the previously mentioned fact that only three of the four FCRs required for 1-Byzantine failure resilience need be processors. The three processors are required to form a fault masking group for output data consensus, and the fourth FCR is required for correct input data consistency. Given flexibility as to the implementation of the fourth FCR, the issue of reliability was considered. A minimal data exchanger requires less complex hardware than a processor and therefore is less likely to fail. Choosing an interstage design for the fourth FCR thus increases system reliability. The second fundamen-

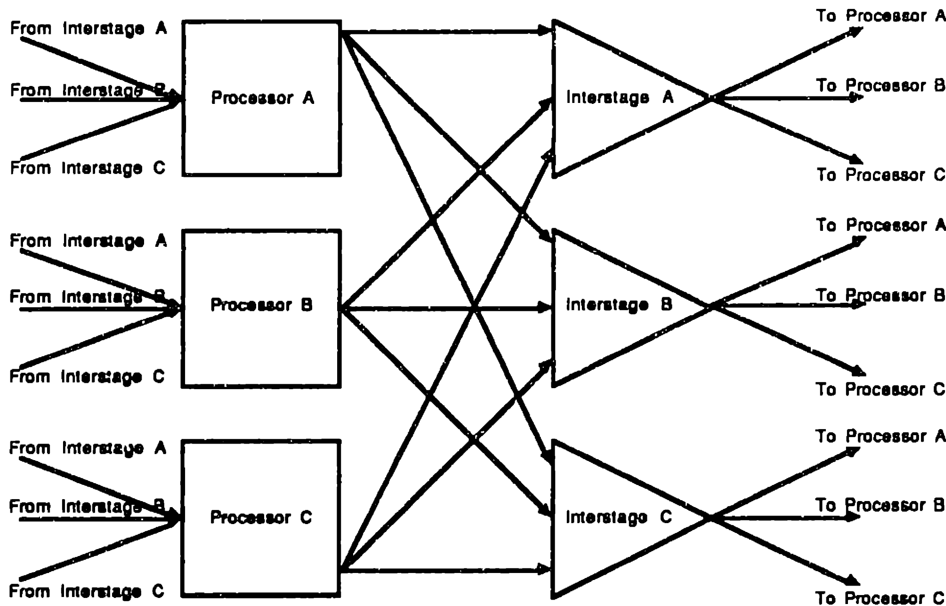


Figure 3.6: AIPS Fault Tolerant Processor (Triplex)

tal difference is that the mechanisms for synchronization and data comparison are provided in hardware in an attempt to reduce the overhead of implementing fault tolerance.

Figure 3.6 illustrates how the FTP satisfies the requirements for cardinality and connectivity. For clarification, the two rounds of the input consistency algorithm occur as follows. In the first round, the source processor forwards its value to all interstages. In the second round the interstages send what they received in the first round to each processor. At this point the processors each have three values which can be resolved to one by a 3-way, maskable, bit-for-bit, majority voter circuit. The important observation must be made that in the presence of an arbitrary failure on behalf of any single FCR (processor or interstage), at most one of the three values received by any processor is different from the others. Agreement will thus be reached. Given that the failure was not in the source processor FCR, then at most one of the three values received by any processor will be incorrect. Agreement and validity are thus ensured.

The FTP is synchronized by employing a digital phase lock loop circuit in each processor to generate a clock signal, called FTC - Fault Tolerant Clock, which is locked in phase in all FCRs. The local version of FTC, henceforth called LFTC, is approximately a 888 kHz clock with a 50 percent duty cycle. Each processor forwards its LFTC to the interstages and the interstages relay them (now denoted ILFTC, for Interstage LFTC) back to the processors. Note that additional connectivity is provided for these LFTC and ILFTC signals. Each processor receives three ILFTCs and generates FTC by selecting the median value. The phase of FTC is compared with the phase of LFTC, and the time until the next rising edge

of LFTC is adjusted depending on the deviation. Regardless of the adjustment, all processors receive the same number of processor clock (SYSCLK) ticks during a FTC period. If a FCR is fast it will wait longer till asserting LFTC and slow the frequency of SYSCLK accordingly so that no extra processor clock ticks are incurred. Synchronization is provided in that a fast processor will have its SYSCLK frequency reduced during the adjustment period relative to a nominal processor. Similarly a slow processor will have its SYSCLK frequency increased relative to a nominal processor. The synchronization scheme imposes a constraint of clock determinacy to guarantee that each processor executes the same number of instructions per FTC cycle. This allows data exchanges to simply be controlled by the FTC cycle. All processors execute the data exchange instruction at the same point in the FTC cycle due to clock determinacy. Hence the data exchange can be controlled by the phases of the FTC.

This is a fine grain synchronization mechanism. Synchronizations occur once each FTC cycle, which corresponds to a rate of 888 kHz. The post-synchronization skew is also very small, approximately $125ns$. This translates directly into performance and reliability improvements over SIFT, as will be illustrated in Chapter 5. However, there are some constraints imposed by this scheme. First is clock determinacy, with its ramifications as previously discussed. Second, there exist some software issues. The fault tolerance of the FTP has been made relatively transparent to the user. The programming model is essentially identical to a simplex Von Neuman machine with some data exchange primitives mixed in. However, the clock determinacy constraint precludes the use of n-version programming techniques in the FTP core itself. A solution to the n-version programming problem has been provided via the use of *attached processors* in each FCR which are not clock deterministic [LA88]. The processors of the FTP achieve what is called *framewise functional sycnhrony.* They are sychronized to FTC frames, and because they are clock deterministic they are performing the same functional operation in each frame.

The data exchange mechanism is elegantly simple due largely to the clock determinacy constraint. A data exchange is performed by each processor during the same FTC cycle. For simplicity all data exchanges are two rounds. The data exchange is pipelined. A byte transaction between processor and interstage occurs every quarter FTC cycle. Data is forwarded to the interstages on the rising edge of FTC and sent back one quarter FTC cycle later. The duplicate copies received at the processor are compared using a maskable, bit-for-bit, majority vote circuit implemented in a programable logic device. The output of this circuit is guaranteed to be valid and readable by the processor at the following falling edge of FTC. Error information called *voter syndrome*, which indicates which processors, if any, were in disagreement during a vote, is accumulated in error registers which are readable by the processor. The steady state throughput of this mechanism is 16 bits per FTC cycle.

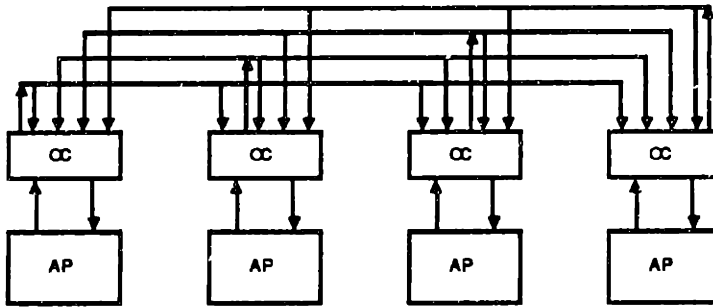


Figure 3.7: MAFT Architecture

3.2.3 The Multicomputer Architecture for Fault Tolerance (MAFT)

The MAFT architecture is another attempt at removing the overhead of fault tolerance from the application domain in order to increase performance and flexibility. The MAFT solution is to provide two entities per FCR, one handling the overhead of fault tolerance, the other running the application. Each MAFT FCR consists of an *operations controller* (OC), and an *applications processor* (AP) [WKF85,KWFT88]. The OC hosts the inter-FCR communication interface, and has two way communication with its AP. The OC is responsible for inter-FCR communication, synchronization, data voting, error detection, task scheduling, and reconfiguration. The AP is responsible for the application task and application dependent peripheral devices. The result is the architecture shown in Figure 3.7.

The MAFT machine consists of "several" (several = 4, Figure 3.7) FCRs fully interconnected via broadcast buses. This configuration satisfies the cardinality and connectivity requirements for 1-Byzantine failure resilience.

MAFT employs a framewise steady state synchronization scheme. A single iteration of this exchange involves broadcasting two versions of a synchronization message. At an a *priori* determined time, the OC executes a "pre-sync" synchronization exchange. The messages exchanged at this time are "time stamped" as they are received. A voted value of the received time stamps is compared with the local pre-sync time stamp and an adjustment is computed. The adjustment affects the time waited until the second version of the synchronization message ("sync") is broadcast. Note that unlike SIFT, the actual time value is not sent, rather reception of the pre-sync message implicitly generates timing information. Additionally, unlike FTP, no extra connectivity need be provided for the timing information; it can be sent over the data broadcast buses. The result is that the FCRs of MAFT are loosely synchronized. The highest frequency of synchronization is 357 Hz [KWFT88]. The maximum skew between FCRs is $18\mu s$ [KWFT88].

Data comparison is performed in a unique manner in the MAFT architecture. The data from the AP which is being exchanged is tagged to identify AP and task. A "data flow"-like voter is implemented which votes values "on the fly". Exact or approximate agreement can be supported by the voter. The effective bandwidth

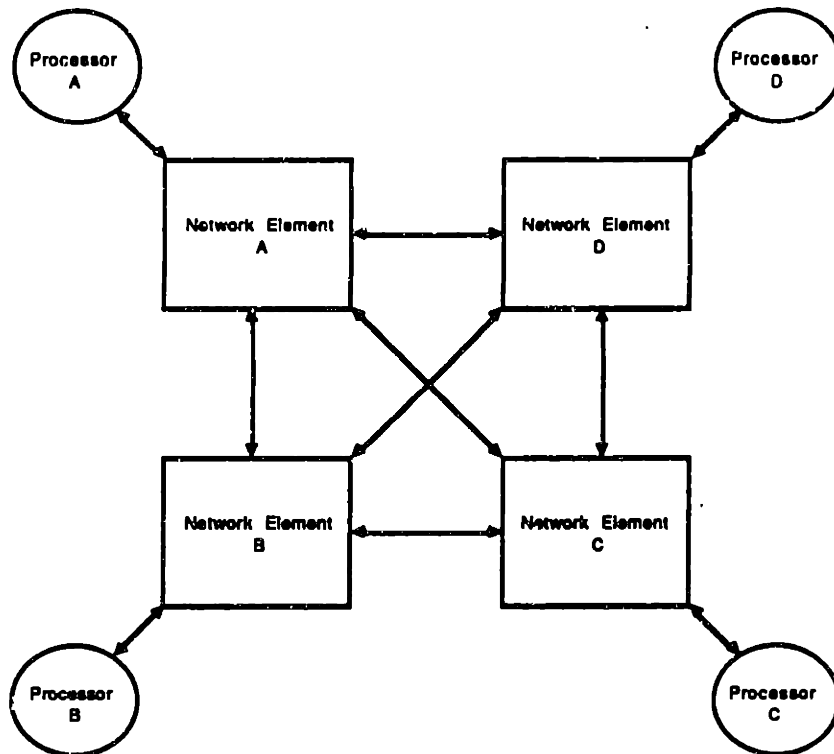


Figure 3.8: NEFTP Architecture

of the data communication mechanisim is quoted at one Mbps (million bits per second) [KWFT88].

It is argued that these aspects of MAFT make it amenable to hardware and software diversity [WKF85,KWFT88]. Implementation of approximate agreement in the voting algorithm removes the bit-for-bit agreement constraint, which is a stumbling block for n-version programming software diversity, and hardware diversity. "On the fly" voting coupled with the synchronization mechanism remove the need for tasks to be run in exact synchrony. However, there is the usual temporal constraint on the execution time of tasks. The programmer must know the maximum possible execution time of a given tasks. The OC function of scheduling is therefore dependent on the application. Furthermore, any software diversity must be accounted for in the scheduler.

3.3 The Network Element Based Fault Tolerant Processor (NEFTP)

A new architecture known as the Network Element Based Fault Tolerant Processor (NEFTP) has been developed (Figure 3.8). It consists of four FCRs connected with fiber optic broadcast links. Figure 3.8 illustrates that the cardinality and connnectivity requirements are satisfied. The design of the NEFTP employs some

of the advantageous features of both the FTP and the MAFT.

The fault containment regions of NEFTP look similar to those of MAFT. Each FCR consists of a *processing element* (PE) and a *network element* (NE). The NE is a hardware implementation of the fault tolerance related functions of synchronization, data communication and data voting. The PE is a commercially available computer which performs the application, scheduling and reconfiguration tasks. The motivation for this architecture is two-fold. First, it removes a significant portion of the burden of fault tolerance from the processor. FTP achieved this by implementing the fault tolerance algorithms in dedicated hardware. MAFT achieved this via the operations controller. NEFTP also mitigates the overhead of fault tolerance by implementing the fault tolerance algorithms with dedicated hardware in the NE. Second, it provides for some flexibility in the implementation (both hardware and software) of the processing element. NEFTP avoids the necessity of clock deterministic processors by providing a means for the NEs to interactively arrive at a consistent view of the data exchange status of the system.

The basic operation of the NEFTP is described here. The PE views the NE as a memory-mapped, buffered I/O device resident on its PE/NE interface. The PE/NE interface in this implementation is Motorola's open architecture VMEbus. The PEs use their NEs to send messages among themselves. These messages are sent for the purposes of synchronization, output consensus, and interactive consistency. If the PE wishes to perform a data exchange, it merely writes the data (message) to its NE. Voted data from the exchange along with error information is eventually available for the PE to read at its NE. The NE appears as a buffered I/O device in that the processor may write many messages to its NE before reading the results from the first exchange. The NEs synchronously perform an interactive consistency exchange of the status of their PE/NE interfaces. Upon arriving at the consensus that a PE message needs to be exchanged, this exchange, either one or two rounds of communication, takes place. A two round exchange is required for input data consistency, but a one round exchange is sufficient to achieve output consistency. Upon completing the exchange, the NEs return to performing their interactive consistency exchange of system status.

The NEs of the NEFTP are synchronized by message receptions. Since message transmission is a *functional* characteristic of this system, this synchronization scheme is known as *functional synchronization*. The NEs are clock deterministic, and are continuously exchanging messages, either the status message or a message from a PE. Each NE notes when it receives messages from the other NEs, and compares the median reception event with the time it sent the corresponding message. Based on the results of this comparison the NE will adjust the amount of time it waits till it starts transmission of the next message. The network elements achieve *functional framewise synchrony*. The NEs are seen to be framewise congruent (like SIFT, FTP, and MAFT) by defining a frame to be a message transmission. A frame is a variable size quantity in the NEFTP because variable message sizes are sup-

ported. The NEs are functionally congruent because they are clock deterministic and the same functional operation is taking place in the same frame on all NEs. The NEs are tightly synchronized because the frequency of synchronization is great, and the post-synchronization skew is small. Synchronization events occur no less frequently than the maximum message (or frame) size which is $30\mu s$ (240 NE data clock periods). The post-synchronization skew is no greater than three NE data clock periods $375ns$. What is the importance of a tightly synchronized NE? One important point is that the granularity of action of the NEs will be a lower bound of the granularity of action of the processors. Also the fact that message transmission is used to synchronize the NEs and that the frequency of synchronization is great, is indicative of the high bandwidth communication mechanism provided by the NE.

The PEs also use reception of messages for synchronization. In order to make reception of a message a synchronizing act for the processor, it must suspend useful computation pending arrival of the message (otherwise known as "busy waiting"). The resultant post synchronization skew is quite small because the messages are made available to the PEs synchronously by the NEs. The post synchronization skew is the NE skew plus the busy wait loop length. A PE need not busy wait for every message, but clearly the frequency with which it does busy wait affects the granularity of action of the system. The natural question to ask is, "... are the PEs, and hence the system as a whole, tightly synchronized or loosely synchronized?" The answer is "... well, it depends." The frequency of synchronization can be varied depending on the application. Applications requiring fine granularity of action can execute the busy wait with each message sent and use the smallest message size available. A mechanism is even provided for the processors to synchronize without actually sending any data in an attempt to maximize the achievable frequency of synchronization. The processor can send a SYNC message which must be enabled by the NE interactive consistency task, but once enabled the exchange consists of the NE writing some status information to the PE/NE interface. Busy waiting for the reception of this status information will indeed be a synchronization event, with the benefit of saving the time involved in sending a minimum length message. Cleary repeated SYNC exchanges would result in the tightest synchronization, but this is not sustainable, as a fault tolerant computer that never exchanges any data will never output any data and is therefore of limited use. An application where a high granularity of action may be desired is in a hardware and software design diversity test. Of course the NE must know the maximum PE skew, hence it is a programmable parameter (called the timeout) of the NE interactive consistency algorithm. The NEFTP sychronization scheme allows the system to tailor the granularity of action to the application.

There exist hardware constraints. The data produced by all PEs must be compatible with bit-for-bit majority voting. The only other constraint is that some common PE/NE interface be used. Currently it is the VMEbus, but could just as easily be any other open bus architecture, standard serial interface, or LAN pro-

tocol. The existing software constraints are also minimal. The current processor model is simplex Von Neuman with two additional primitives SEND and SCOOP[1]. SEND enqueues a message for transmission in the PE/NE interface. If it is not possible to enqueue a message because the NE is full, SEND will read messages from the NE until it becomes clear to send. SCOOP will read all messages that have been sent since the last SCOOP, busy waiting if data is not available at the NE. SCOOP will then send a synchronization message and wait for its reception. Thus upon completion of a SCOOP, all messages that have been sent prior to calling SCOOP have been received and the PEs are synchronized. Tailoring the granularity of action of the NEFTP simply involves programming the NE with a new timeout and perhaps adjusting the ratio of SENDs to SCOOPs. For example, to make the finest grain NEFTP set the NE timeout to the minimum value, and call SCOOP after every SEND.

The data exchange mechanism provides for one and two round exchanges. The results are voted by a 4-way, maskable, bit-for-bit, majority vote algorithm implemented in a programmable logic device. The mechanism is ensured of voting the redundant copies of the same data because the NE preserves the total ordering of messages. The total ordering is preserved because of two reasons. First, the buffering in both directions at the PE/NE interface is implemented with *first-in-first-out* memories (FIFOs). Second, the NEs continuously perform an interactive consistency exchange of the status of their PE/NE interfaces; and they only enable a PE message for exchange when all particpating PEs request the message, or when the timeout period elapses after a majority of participating PEs request the message. This ensures that a faulty PE cannot cause the total ordering of messages to be violated as viewed by non-faulty PEs. Unlike the FTP not all exchanges take two rounds. Regardless of the exchange, the completion of the exchange results in all NEs having sufficient redundant copies of the data to satisfy agreement by voting.

The NEFTP has been shown to satisfy the requirements for 1-Byzantine failure resilience. It does so with few constraints on other aspects of the architecture. The architecture can be tailored to be tightly synchronized for applications where fine granularity of action and high performance are required. Granularity of action can be traded for design diversity in processing hardware and software. This will have a corresponding detrimental impact on performance, however the impact on reliability is less clear. Larger granularity of action will tend to lessen reliability, but this may be compensated by increased resilience to common mode design failures in the processing hardware and software. The NEFTP also incorporates sound fault containment region design. The inter-FCR communication network employs fiber optic links which are quite well suited for fault tolerant applications. No other media has better isolation and attenuation properties. Fault containment regions can finally be truly isolated, both electrically and physically.

A prototype NEFTP has been built as part of this research. In the prototye the

[1]These message exchange primitives have been designed and implemented by Steven A. Friend

processing elements are commercially available 32 bit, 68020 based, single board computers. The design of the network element is presented in Chapter 4.

Chapter 4
NEFTP: The Network Element Design

The Network Element (NE) is a hardware implementation of the fault tolerance related functions of synchronization, data communication, and data voting. The NE can be viewed as an abstract communication network. This abstract view of the NE is a simple case of the Byzantine Resilient Virtual Circuit Abstraction (BRVC) [Har87] The BRVC (Figure 4.1) has several interesting properties. It preserves the total ordering and integrity of all valid messages. It provides the resources to correctly manage sufficient redundancy for one and two round consensus exchanges, and allows for graceful degradation of this redundancy (i.e. it is Byzantine Resilient). The BRVC network is synchronizing. This means that if the input satisfies the property that redundant copies of the input message arrive at the network within some known t_{skewin} then this message will be output at all processors within a known $t_{skewout}$: ($t_{skewout\ max} < t_{skewin\ max}$). A desirable feature of such a network is that it have very high bandwidth so as to not degrade the throughput of the aggregate system. Also, it must be testable. The NE has been designed to satisfy these properties.

4.1 The Basic NE cycle

In order to fulfill the network abstraction, the NE executes the basic cycle shown in Figure 4.2.

The processors of the NEFTP communicate via message exchanges. Message exchanges are used for the purposes of synchronization, output consensus, and input consistency. Each NE must tell all other NEs whether its associated processor has an exchange request pending. The NEs will then have sufficient information to arrive at a consensus regarding exchange requests. If an exchange request is valid (i.e. all non-faulty processors have requested the exchange), then the exchange will be honored. All NEs will perform the processor exchange and then return to exchanging processor requests. Essentially four input consistency exchanges occur in the

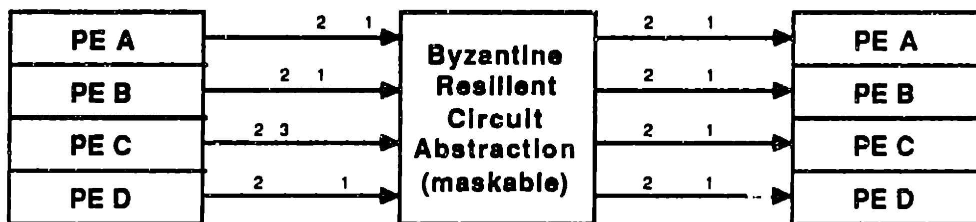


Figure 4.1: BRVC Abstraction

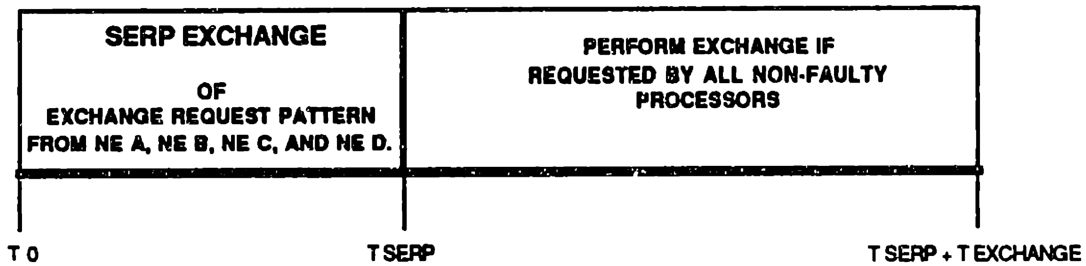


Figure 4.2: Basic NE Cycle

first half of this cycle. The data being input is an *exchange request pattern*. This sequence of four input consistency exchanges has been somewhat optimized and masquerades under the title *system exchange request pattern* (SERP[1]) exchange. By executing the SERP exchange the NEs interactively arrive at a Byzantine resilient consistent view of the PEs' exchange requests, from which each NE can make the same decision about whether to perform the exchange. This SERP exchange is fundamental to BRVC abstraction discussed above. It is what allows the NE aggregate to be synchronizing and maintain total ordering of message transactions without imposing the constraint of clock determinacy on the processors.

4.2 The NE Functional Sub-Sections

The NE can be considered to be composed of six sub-sections. They are the following:

1. The processor/network element interface.
2. The network element data paths.
3. The inter-FCR communication links.
4. The network element fault tolerant clock.
5. The network element scoreboard.
6. The network element controller.

The function performed by each subsection is depicted by its name (with the exception of "scoreboard"). The functionality of each subsection will be made clear below. Before beginning, some discussion of nomenclature is necessary. In general, names in uppercase letters refer to a network element signal (to be found on the schematic) or some acronym for a functional module of the NE (such as itself). Names of NE functional modules which are not acronyms will be introduced in *slanted text*. Also, signal names ending in "*" or "/" denote a low true signal.

[1]SERP concept and nomenclature courtesy of Stuart J. Adams.

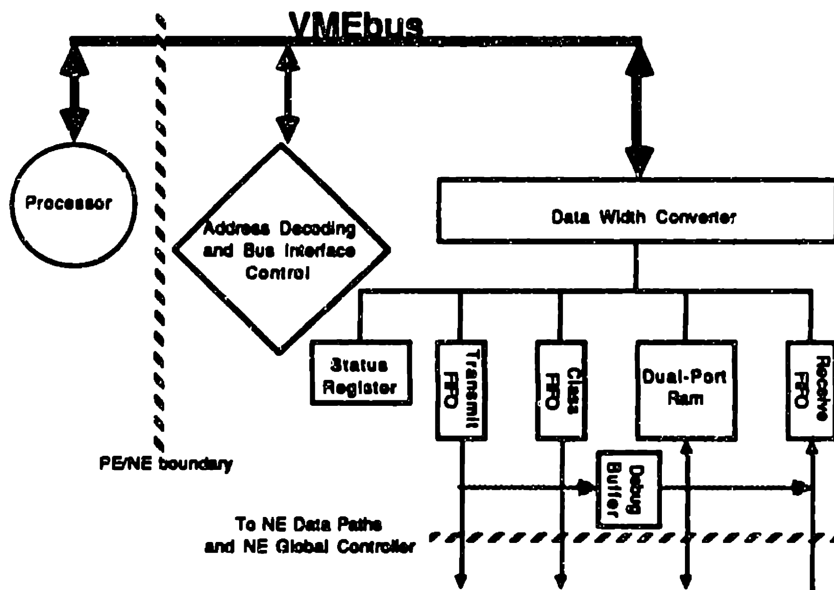


Figure 4.3: PE/NE Interface Block Diagram

4.2.1 The PE/NE Interface

The PE/NE interface must provide for the efficient and ordered passing of variable size messages between the PE and the NE. It must also insure that no valid messages get overwritten.

The interface implementation is based on the industry standard VMEbus by Motorola. The VMEbus was chosen for several reasons. As an industry standard, it does not seriously constrain the choice of PE implementation. It also has reasonably high bandwith, particularly considering the fact that it only hosts one processor in the NEFTP configuration (Figure 4.3).

In the NEFTP configuration, the PE is a commercially available, 32 bit, single board computer which acts as bus master. The NE is a bus slave. Specifically, the NE is an *A16*, *D32* slave. *A16* implies that it is addressed with 16 bits, and *D32* implies that it supports data transfers as wide as a full *longword* (32 bits). In fact the NE can be viewed as a slave that has exactly 5 longword ports (Figure 4.3):

- a transmit FIFO
- a receive FIFO
- a class FIFO
- a status register
- a dual-ported RAM.

All of these ports are accessed via longword transactions by the processor; however none of these ports are actually wider than 8 data bits, and only two of them actually use all 32 bits of data. Therefore, all 5 ports are attached to the bus via a 32 bit *data width converter* (Figure 4.3). The data width converter allows the processor to deal only in longwords yet communicate with 8 bit wide ports. This decision was made for several reasons. A narrow port implementation was chosen to preserve real-estate on the NE board which is a 6U by 160 VME compatible Eurocard[2]. Longword transactions were chosen to maximize communication bandwidth between the NE and PE for critical transactions. Longword transactions were chosen for the non-critical ports to provide for unified and simplified bus interface control (Figure 4.3). The utilization of these ports is described below.

The processor writes messages to be exchanged with other processors to the transmit FIFO (XMIT FIFO). It is conceptually a write only, longword port. It is actually implemented with a 2K (2K = 2048) by 8 FIFO memory. It is a critical port because the rate at which the PE can write messages to the NE affects the overall NEFTP throughput; therefore, it is one of the two ports that uses all 32 bits of data. A write to the XMIT FIFO stores a longword in the data width converter. The data is then transfered to the XMIT FIFO by four sequential byte write operations.

The processor reads the results of a message exchange with the other processors from the receive FIFO (REC FIFO). It is conceptually a read only, longword port, and is again implemented with a 2k by 8 FIFO memory. It is the second critical port and thus uses all 32 bits of data. A read of the REC FIFO causes four sequential byte reads of the FIFO memory to occur. These four bytes are packed in the data width converter and then presented to the processor as a longword.

The processor writes its exchange request pattern, used in the SERP exchange, to the class FIFO. The class FIFO is conceptually a write only longword port of which only the least significant seven bits are used. It is physically a 64 by 8 FIFO memory. The exchange request pattern is written after the corresponding message has been written to the XMIT FIFO. Writing the exchange request pattern signals the NE that a message is resident in the XMIT FIFO. The value of the exchange request pattern denotes the size and type (or class, hence the name class FIFO) of the message. Four bits are used to denote the size of the message, and three bits are used to denote the exchange type. There are currently seven exchange types implemented in the NEFTP (Figure 4.4).

The *SYNC* exchange involves sending no data and is the minimal overhead synchronizing act of the NE. The *FromV* and *FromV with masks* exchanges, are single round output consensus exchanges. The latter additionally outputs the new system configuration, or mask, byte to the NEs. The mask byte informs the NE which NEs and PEs in the system are thought to be non-faulty. The remaining four (*FromA, FromB, FromC, and FromD*) exchanges are the four possible input

[2]6U by 160 is one of the available sizes, perhaps the standard, for VMEbus cards.

Exchange Request Pattern Bits 6 5 4	Exchange Type
0 0 0	Not currently used.
0 0 1	SYNC. No data exchanged. NE error information latched at REC FIFO.
0 1 0	FromV. (From Vote). Output consensus exchange.
0 1 1	FromV with masks. Output consensus exchange. NE configuration, or mask, byte is output to NE.
1 0 0	FromA. Input consistency exchange from FCR A.
1 0 1	FromB. Input consistency exchange from FCR B.
1 1 0	FromC. Input consistency exchange from FCR C.
1 1 1	FromD. Input consistency exchange from FCR D.

Figure 4.4: NE Exchange Types

consistency exchanges. Each of these is a two round exchange.

All exchange types, except *SYNC* can accept messages of any allowable length (in bytes) other than zero. The *SYNC* exchange is performed as part of the SERP exchange that enables it. The current size decoding scheme is to simply multiply the value of the exchange request pattern size field by 16.

The status register provides the processor with flow control status of the PE/NE interface along with the NE identification (NEID). The status register is conceptually a read only, longword port. It is physically a 4 bit buffer, two bits for flow control information, and two bits for NEID. The flow control allows the processor to write several messages prior to reading any, or to block attempting to read a message it has just sent, without destroying any valid messages. Therefore one bit (CTS, for Clear To Send) is required to tell the processor that a message can be written to the XMIT FIFO, and another bit (DATAREADY) is required to tell the processor that a message has arrived and can be read at the REC FIFO. CTS is set whenever the XMIT FIFO is less than half full and the class FIFO is not full. This conservatively insures that a maximum size packet can be successfully written to the NE. DATAREADY is set whenever the REC FIFO is not empty. This insures that a message is currently resident in the REC FIFO, or at least being delivered by the NE to the REC FIFO. In either case the processor will be able to read the message successfully. The simplicity of this flow control mechanism implies that the processor's data exchange primitives must maintain ordered queues of the size

of messages that have been written to the NE in order to know what size message to read from the REC FIFO. The NEID field is driven by a two bit dip switch on the NE which identifies the FCR.

The dual-ported RAM provides a debug interface between the processor and the network element. It is conceptually a read/write longword port and is physically a 2K by 8 dual-ported memory that arbitrates contention for a single location via a BUSY flag. A NEFTP Interactive Debugger[3] has been constructed around this dual-port interface. Upon power up, the PE and NE enter into debug mode. The PE begins executing code which allows the user to run tests of the NEFTP. The primitives of these tests are the commands of the Interactive Debug Command Language which the processor can write to a location (the command register) in the dual-port memory. In debug mode, the NE loops on the command register for a debug command. Upon detecting a valid command, it executes the specified routine, zeros the command register and loops for the next command. In this manner, the user can exercise all facets of the NE hardware from a terminal. The Interactive Debugger Command Language commands can also serve as the primitives for any automatic NEFTP self test and diagnostic code. Aside from the command register, dual-port memory locations are also used to specify various parameters of the NE that are used in debug modes. The reader may wonder how the NE can be clock deterministic with this dual-port. Since the PE/NE interface is asynchronous, at any given time, one NE's dual-port may be BUSY while another's is not. Actually the NEs need only be clock deterministic while they are operating in the redundant fault tolerant configuration. So if the dual-port is strictly used in debug mode then this is not a problem. However, it is desired not to constrain any future uses of the dual-port. Therefore, the NE dual-port interface is made clock deterministic by forcing the worst case wait with every NE access to the dual-port.

The processor's memory map of the NE is shown in Figure 4.5. A simple address decoding scheme was desired. The NE must decode at least 3 address bits to uniquely access its five ports. A memory block size of 512 longwords was chosen. This allows the entire 2K by 8 byte dual-port to be uniquely addressed by longword transactions in 4 blocks. There are 4 blocks left over for the other 4 ports that comprise the NE. Therefore, only three bits need be used for address decoding with minimal waste of address space.

The schematics for the PE/NE interface are presented in Appendix A and will be refered to by sheet number in the following discussion. Many of the schematics include programmable logic devices (PLDs). The logic equations implemented in a given PLD are included in the corresponding "file" in Appendix B. However, the discussion below is intended to be understandable without reference to the Appendix B.

[3]The NEFTP Interactive Debugger is essentially cloned in concept, and structure of implementation, from a similar interactive debugger developed for the CSDL Fault Tolerant Parallel Processor by Stuart J. Adams.

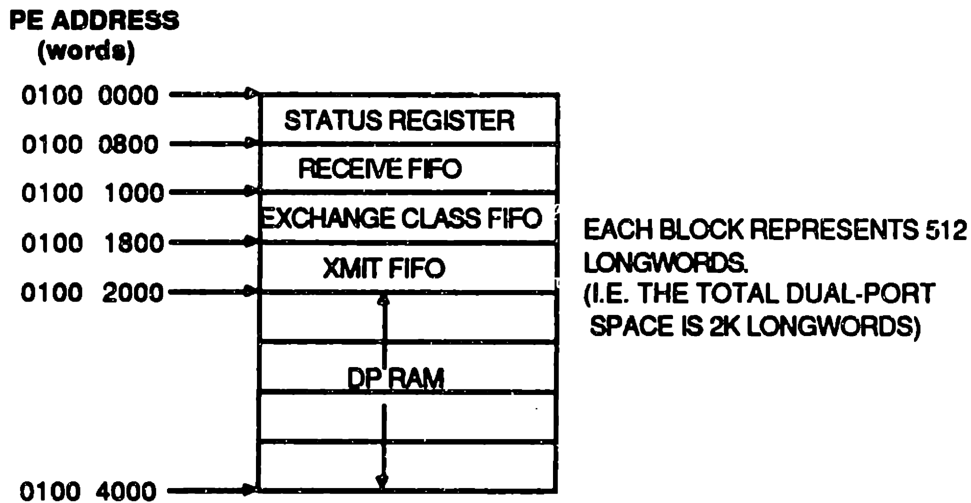


Figure 4.5: NE Memory Map

Sheet 1 illustrates the schematic for address decoding and bus interface control. To understand the terminology used in this schematic, one must understand how the basic VMEbus read and write cycles are performed with the NE.

A read cycle has the following format. The processor presents the address and address modifiers to the slave, in this case the NE. The processor then asserts its control lines (LWORD*,AS*,DS0*,DS1*, and WRITE*) to specify the kind of cycle. For a read of any port of the NE all these are asserted except WRITE*. A valid NE address causes the *address decoder* to assert BOARD SEL/ . An active BOARD SEL/, with the appropriate control signals causes the *data strobe generator* to assert the data strobe for the correct port as decoded by A11-A13. This starts the access of the data from the selected port. It also causes the *DTACK* (Data Transfer ACKnowledge) *generator* to assert DTACK after a delay that ensures valid data is presented to the processor first. The data strobe generator also asserts BOARDG/ which causes the bi-directional data width converter (sheet 3) to be turned on in the direction appropriate for a read. When the processor sees DTACK asserted, it can read the data and terminate the access by de-asserting the control signals.

A write cycle works in much the same way as a read cycle with some exceptions. The processor presents the address, address modifiers and the data to the NE. The processor then asserts all of the control signals. BOARD SEL/, BOARDG/, and the appropriate data strobe are asserted. The data strobe generator asserts another signal called CAB (Clock A to B) which immediately registers the data at the data width converter (sheet 3) from the processor side (A) to the NE side (B). DTACK will be generated after the NE has stored the data at the appropriate port. The processor can then terminate the access.

Sheet 1 contains another device (U0105) which is used as a latch for A07-A02 as well as a counter for block transfer capability. According to VMEbus specifications,

a slave must locally increment the initially accessed address during a block transfer. Block transfers cannot cross 256 word boundaries, so the NE need only increment A7-A2. Block transfers are currently not used, but are implemented to provide a means of future performance enhancement.

Sheet 2 shows 3 of the 5 ports that comprise the NE. These are the dual-port RAM, the class FIFO, and the status register. All of the NE ports are attached to the FIFO DATA bus which comprises the NE side of the data width converter.

Sheet 3 shows the data width converter along with the XMIT and REC FIFOs. Also, there are two control PLDs (U1413 and U1417) which provide the control signals to sequence the XMIT and REC FIFOs. When the XMIT FIFO is written to, a longword is latched at the data width converter. RXMITFIFODS is asserted by the data strobe generator. RXMITFIFODS causes U1417 to assert STARTXSHIFT and begin shifting a byte at a time out of the data width converter and into the XMIT FIFO. When all four such shifts have been completed WRDONE is asserted to signal the DTACK generator that it is time to assert DTACK. A similar process occurs on a read of the REC FIFO. The interesting thing to notice is that due to the simplified PE/NE interface flow control scheme, it may be possible for the REC FIFO to become empty during a read of a single longword. This is possible because the NE loads the REC FIFO one byte at a time, and the PE which reads longwords only looks at the FIFO empty flag. If this were to happen, the NE must be in the process of loading the REC FIFO, and the read would be delayed until the data was available. However, this operation would be transparent to the PE because the NE would be able to write 4 bytes to the REC FIFO before a bus error would result from a timeout. This will never be a problem, regardless of the relative speeds of the PE and NE so long as the bus timeout period is longer than three NE byte times.

The VME interface subsection connects with the data paths subsection via two buses. Data is sent into the NE data paths from the XMIT FIFO via the SOURCE DATA bus; and data is delivered from the NE data paths' voter to the REC FIFO via the VOTED DATA bus. There exists a debug path which connects the XMIT FIFO output with the REC FIFO input (Figure 4.3). This allows the PE/NE interface to be somewhat isolated from the rest of the NE for testability.

4.2.2 The NE Data Paths

The NE data paths provide the resources to handle one and two round communication exchanges, as well as the ability to resolve redundant data copies. To provide this functionality the data paths of each NE include the following elements (Figure 4.6):

- A link which connects the XMIT FIFO to the inter-FCR communication link transmitter (the MY EXTERNAL DATA bus). This allows the NE to broadcast messages from its PE, through the XMIT FIFO, to the other NEs.

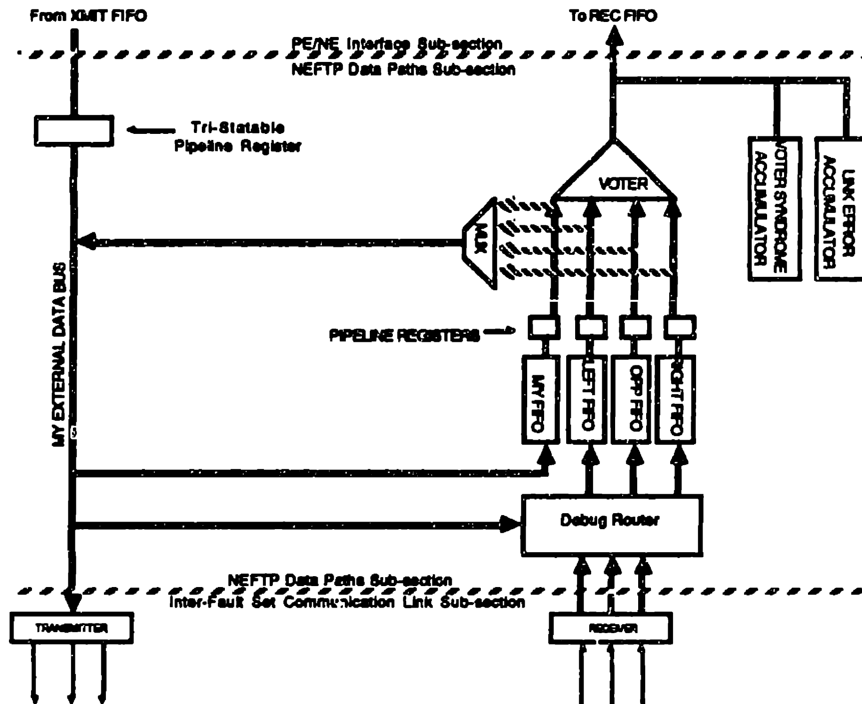


Figure 4.6: NE Data Paths Block Diagram

- Intermediate storage for messages received from the other three NEs. This storage is implemented in 3 FIFO memories, the LEFT, OPP, and RIGHT FIFOs. More will be said about this nomenclature below.

- Intermediate storage for copies of messages sent by their local transmitters. This storage is implemented in a FIFO memory, the MY FIFO.

- A means of rebroadcasting a message from any of the intermediate storage FIFOs. This is performed by a multiplexor (MUX) which can route any of the intermediate FIFO outputs to the NE's transmitter.

- A PLD implementation of a 4-way, bit-for-bit, maskable, majority voter which resolves redundant copies of a message in the intermediate storage FIFOs to a single message.

- Error accumulators that record the occurrence of any transmission errors detected in receiving a message from another FCR, or comparison errors detected in voting the redundant copies of a message, These errors are accumulated over the duration of the message as *link error*, and *voter syndrome* respectively. The values of these accumulators are appended to each message as it is delivered to the REC FIFO.

- A *debug router* which allows any of the external (i.e. LEFT, OPP, RIGHT) intermediate storage FIFOs to be driven by the MY EXTERNAL DATA bus.

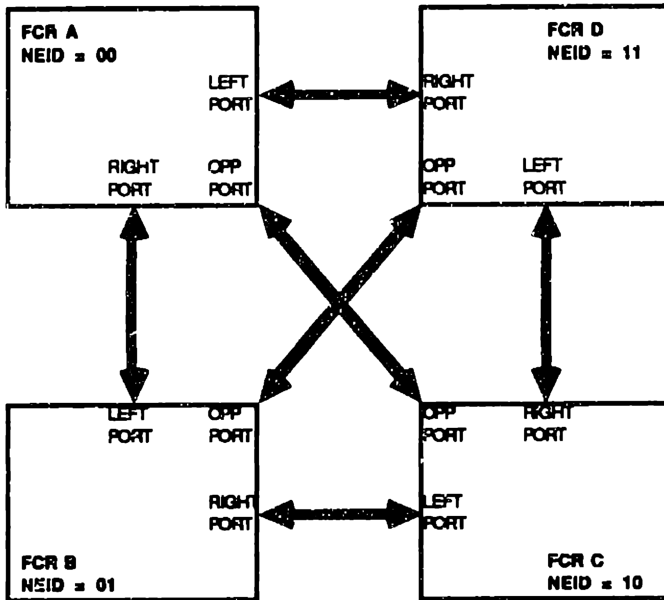


Figure 4.7: NE Physical and Virtual Identifiers

Testability of the data paths is thereby provided by allowing the data paths to be isolated from the inter-FCR communication links.

The interaction of the various NE data path elements will be illustrated in the explanations of one, two round and SERP exchanges below. Before continuing, some mention must be made of the correlation between names identifying FCRs as A, B, C, and D, and names identifying FCRs as MY, LEFT, OPP, and RIGHT. This discussion is aided by a diagram of the FCR identifiers (Figure 4.7). The former group of names are the *virtual* FCR identifiers. The latter group of names are the *physical* FCR identifiers. The virtual identifiers remain constant across FCRs and are set by the NEID switch (i.e. FCR A means identically the same thing to all FCRs). Therefore the virtual identifiers are used by all processors so as to make coding of the data communication primitives and FDIR primitives independent of physical FCR. The physical identifiers are not constant across FCRs (e.g. FCR D is the LEFT FCR as viewed by FCR A, whereas it is the RIGHT FCR as viewed by FCR C). The physical identifiers are necessary to correctly track and route messages through the NE data paths. Therefore, the NE must translate between virtual and physical identifiers based on the value of its NEID. In the discussion below, this translation will often be left up to the reader; a clear understanding of Figure 4.7 should make this task trivial.

In a one round exchange, each fault containment region sources a copy of the message (e.g. the result of a computation). Each FCR sends its source message to its transmitter which then broadcasts this message to all other FCRs. Each FCR records what it sent in its MY FIFO while it receives a copy of the message from

each of the other FCRs. Each FCR then stores these copies in its LEFT, RIGHT, and OPP FIFOs. Any link errors that are detected in this process are recorded by the *link error accumulator*. At the end of this single round, four copies of the message are resident in the four intermediate storage FIFOs. These copies are then passed through the byte-wide voter which reduces them to a single copy and masks out any single error. Any errors detected by the voter are recorded by the *voter syndrome accumulator*. The output of the voter is then delivered to the REC FIFO where it is available for the processor to read. The information from the error accumulators is appended to the voter output as it is delivered.

In a two round exchange, only a single fault containment region sources valid data (e.g. an input or interactive consistency exchange). The sourcing NE sends the message from its XMIT FIFO to its transmitter, and records a copy of the message in its MY FIFO. All other NEs source, and record in their MY FIFOs, a null packet equal to the valid message size. At the end of the first round each NE has a single valid copy of the message in one of its intermediate storage FIFOs, and null packets in the others. In the second round, the contents of all intermediate storage FIFOs are output to the MUX. The MUX selects the valid data and routes it to the transmitter to be rebroadcast. Therefore, at the end of the second communication round each NE has four copies of the original message in its intermediate FIFOs. All NEs can now vote and deliver the redundant copies of the message as in the single round exchange. However, to satisfy the input consistency requirements, all NEs must not vote the copy of the message which they received in the second round from the original source FCR [PSL80]. Otherwise the sourcing FCR would have a double influence in the resulting four way vote and could corrupt the output. For example, if the particular exchange is a *FromA*, NE A must mask out the message resident in its MY FIFO at the end of the second communication round. Correspondingly, NE B must mask out the message in its LEFT FIFO at the end of the second communication round, and so on.

A *SERP* exchange is an optimized sequence of four two round exchanges of a single byte. The "first round" consists of each FCR broadcasting its single exchange request pattern as read from the class FIFO. The NEs wait until they have received all four of these bytes; then the "second round" begins. During the second round these bytes are rebroadcast in an ordered sequence. The byte that was initially from FCR A is rebroadcast first, then the byte from FCR B, and so on. At the end of the second round, each NE will have four copies of each exchange request byte, and can vote them to arrive at a consistent value for each exchange request byte.

Sheets 5, 6 and 7 of Appendix A represent the NE data paths schematics. Sheet 5 shows the intermediate storage FIFOs. The input of the MY FIFO is driven by the MY EXTERNAL DATA bus, thereby allowing it to capture anything sent to the transmitter. The LEFT, OPP, and RIGHT FIFOs are driven by the outputs of the debug router. The outputs of each FIFO pass through a pipeline register (sheet 5). These registers are necessary to satisfy the various data path timing constraints.

The outputs of the pipeline registers feed both the voter (sheet 5) and the MUX (sheet 6).

The voter is conceptually a four channel, byte wide, maskable, bit-for-bit majority voter. If a given bit is asserted by any two enabled (non-masked out) channels, then the corresponding bit of the voter output (the VOTED DATA bus) is asserted. There are four mask inputs (the VMASK signals), one for each channel. A mask value of 1 indicates that the corresponding channel's input is used in computing the voted output. A mask value of 0 indicates that the corresponding channel's input is ignored in the voted output computation. More will be said about the mask signals when discussing the NE controller.

Voter syndrome is generated with each byte that is clocked through the voter. There are conceptually five voter syndrome bits:

- LERR. If this bit is asserted, the LEFT channel is in disagreement with the voted value in at least one bit position.
- OERR. If this bit is asserted, the OPP channel is in disagreement with the voted value in at least one bit position.
- RERR. If this bit is asserted, the RIGHT channel is in disagreement with the voted value in at least one bit position.
- MERR. If this bit is asserted, the MY channel is in disagreement with the voted value in at least one bit position.
- QERR. If this bit is asserted, there exists a two-two split over the value of at least one bit. This actually represents a failure of the system in that the validity of the voted output is no longer guaranteed. A QERR is therefore the result of two simultaneous errors on at least one single bit value. It is possible for all the other error bits to be asserted without causing a QERR, if no two discrepancies occurred over the value of the same bit.

The mask values do not affect the syndrome generation. This is important when considering how a faulty FCR is recovered by the non-faulty FCRs. All non-faulty FCRs can tell when a faulty channel has stopped producing voter errors without subjecting their voted output values to corruption.

The voter is acutally implemented as two 4-bit, pipelined "slices" via PLDs. The pipeline is one stage deep. The output is the voted value of the previous input, while the syndrome corresponds to the current input. The majority voting scheme requires that at least two channels must be enabled to assert any outputs. However, the actual implementation allows the MY channel or the OPPOSITE channel to decide the voted output alone if all other channels are masked out. Each slice generates its own syndrome information. The conceptual QERR bit has been broken up into three bits per slice due to lack of product terms in the PLDs.

The seven voter syndrome bits must be amalgamated somehow. This is accomplished via the syndrome accumulator (sheet 7). The sydrome accumulator produces five bits of output, corresponding to the five conceptual syndrome bits presented above, which can be appended to the voted message output via the VOTED DATA bus. Accumulation involves logically "ORing" the values of the five syndrome bits produced for each byte of the message. When the accumulator is read at the end of voting a message, its values are cleared.

Sheet 6 shows the rest of the NE data paths. The MUX, shown on the left, consists of four 2-bit, 4:1 multiplexors. The MUX allows the output of any one intermediate storage FIFO to be selected for *reflection*. Reflection is the name given to the second round of communication of an input consistency exchange. During reflection, one of the intermediate storage FIFOs on each NE holds a message that needs to be rebroadcast to the other NEs. The MUX provides the capability for routing this message to the transmitter via the MY EXTERNAL DATA bus. In the upper left hand corner of sheet 6 there is a register which is used to connect the output of the XMIT FIFO to the MY EXTERNAL DATA bus. This register provides the correct pipeline staging and the tri-state capability necessary to have source messages and reflected messages both drive the transmitter in an identical fashion. The center of sheet 6 shows part of the inter-FCR communication link and shall be discussed later. The debug router is shown on the right hand side of sheet 6. It is also implemented as four 2-bit slices in four PLDs. The debug router provides the same basic switching capability for each of inputs to the three external (i.e. LEFT, RIGHT, and OPP) intermediate storage FIFOs. The intermediate storage FIFO input can be driven by the outputs of the corresponding inter-FCR communication receiver (normal operation), or it can be driven by the MY EXTERNAL DATA bus (debug operation). The operational mode is selected by the debug enable signal (LDBEN, ODBEN, or RDBEN). This provides the capability to test the data paths of a single NE alone.

The intricate control of the NE data paths is somewhat isolated from the NE global controller by the data path controller circuit (sheet 7). This local control of the data paths is divided into two parts.

The *synchronous controller* controls those signals that occur synchronously with respect to the local NE. These involve shifting data out of the intermediate storage FIFOs, operating the voter, syndrome accumulator, and MUX, and shifting data into the MY FIFO. The synchronous controller is implemented in a PLD. The basic operation of this controller is as follows. The NE global controller specifies the function to be performed during the next message frame. This NE function tells the synchronous controller what operation to perform (sheet 7) during the next message frame. In the NEFTP a frame denotes a message transmission slot and is therefore variable depending on the message size. To perform a single round exchange, or the first round of a two round exchange, the function selects are set to 1, and reflection is disabled (i.e. the global controller sets REFENABLE to 0).

This causes data to be loaded into all intermediate storage FIFOs. To perform the second round of a two round exchange (i.e. reflection), the function selects are set to 2. Data is shifted out of, and into all intermediate storage FIFOs. The global controller asserts REFENABLE and presents the synchronous controller with the virtual ID of the FIFO which has the valid message. This virtual ID is translated by the NEID to generate the physical ID. The MUX selects the channel to reflect from based on this physical ID. Setting the function selects at 3 results in the execution of the SERP exchange. More will be said about this when the scoreboard is discussed. The NE function selects do not provide for the voting and delivery operation of the data paths. Voting is controlled by the global controller's VOTEIN signal. Asserting VOTEIN instructs the synchronous controller to shift data out of all intermediate storage FIFOs and into the pipeline registers. To account for the two stages of pipelining between the FIFO and voter outputs, VOTEIN creates two more signals VOTEOUT1, and VOTEOUT which correspond to VOTEIN delayed 1 and 2 data clock periods, respectively. VOTEOUT1 signals valid data in the voter and is used by the syndrome accumulator to denote when the voter syndrome is valid. VOTEOUT denotes when valid data is output from the voter, and is used to enable the voter output onto the VOTED DATA bus.

The *asynchronous controller* (sheet 7) controls those signals which are asynchronous with respect to the local NE's clock. Essentially, it is responsible for loading data into the LEFT, RIGHT, and OPP FIFOs. It also performs the clearing of all the intermediate storage FIFOs. In normal operation the data shifted into the LEFT, RIGHT, and OPP FIFOs comes from the inter-FCR communication receivers, and the clock used to shift the data in is recovered from the same receivers. However, when an external link is put into debug operation, the asynchronous controller must use the local data clock to shift in the data from the MY EXTERNAL DATA bus. To clear a given intermediate storage FIFO, the global controller selects the FIFO with the CLEARDATAPSEL signals and then asserts the CLEARDATAP. The asynchronous controller decodes the clear selects and clears the selected FIFO as soon as all of its shift in activity has stopped.

Sheet 7 also shows the link error accumulator. Each inter-FCR communication link has a violation flag associated with it that indicates that the communication protocol has been violated. The link error accumulator takes these flags as inputs and accumulates them in a manner identical to the syndrome accumulator. The three accumulated results are appended to the delivery of voted data. Reading the accumulator causes it to be cleared. The accumulated link error and voter syndrome are combined to form the first of two NE error bytes that are appended to each message delivery. The signal ERROR1READ enables this byte onto the VOTED DATA bus. Combined in the same PLD as the link error accumulator is a scale counter (sheet 7) that is used to count the TIMEOUT used in the scoreboard. This will be discussed in greater detail in the scoreboard sub-section.

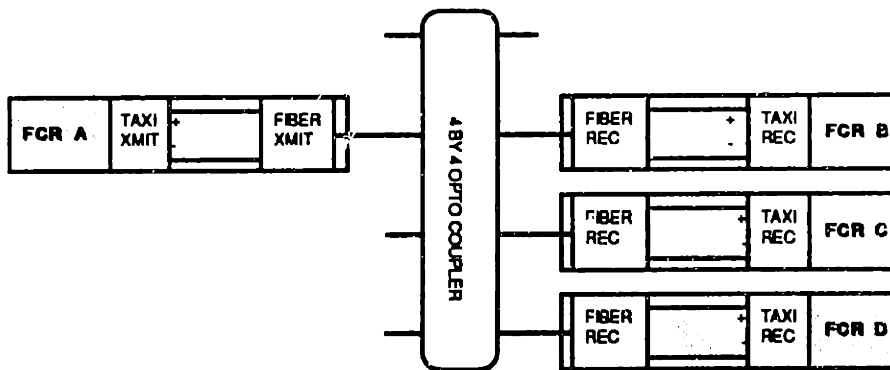


Figure 4.8: Inter-FCR Communication Link Block Diagram

4.2.3 Inter-FCR Communication Links

Each inter-FCR communication link consists of a transmitter broadcast circuit and three receivers (Figure 4.8). This design uses the TAXI chip set by Advanced Micro Devices Co. (sheet 6, center) to interface to a fiber optic broadcast link. The TAXI chipset is an efficient implementation of a high speed, encoded, serial data link with a byte wide parallel interface to the user.

To send data, TTL level parallel data are strobed into the transmitter, encoded into a ten bit word (in the current configuration) and output serially as complementary ECL signals. The ECL outputs drive a 200 Mbps (maximum) fiber optic transmitter (sheet 11). The resulting optical signal is sent to the other three FCRs via a 4 by 4 optical coupler (Figure 4.8). Each of the three receiver circuits converts the optical output of the coupler to an ECL serial bit stream via a fiber optic receiver. The maximum data rate of the fiber optic receiver is 200 (Mbps). These ECL levels are input to a TAXI receiver chip. The encoding scheme is sufficient for the TAXI receiver to recover a timing reference, sample the bit stream, and package the output into a byte. The presence of a valid data byte is signaled by a data strobe. If the receiver discovers an erroneous bit pattern it will assert its violation pin to indicate that the current output byte may be invalid.

Both the TAXI transmitter and receiver must be supplied with a reference byte frequency. If no data is strobed into the transmitter during a byte time the transmitter will send a pre-determined synchronization byte to keep the clock recovery ciruit of the receiver locked in phase. The reception of this *sync* byte is denoted by assertion of the command strobe by the receiver.

This inter-FCR communication link has several desirable properties. It provides excellent isolation between FCRs, both physically and electrically [Sta85]. It is high bandwidth, the maximum possible byte frequency being 12.5 MHz. As a serial link, it minimizes the number of wires between FCRs. To illustrate this point consider that the NEFTP has only 12 inter-FCR links whereas the AIPS FTP has 96 in a typical configuration [GADS88]. This arguably makes for a more

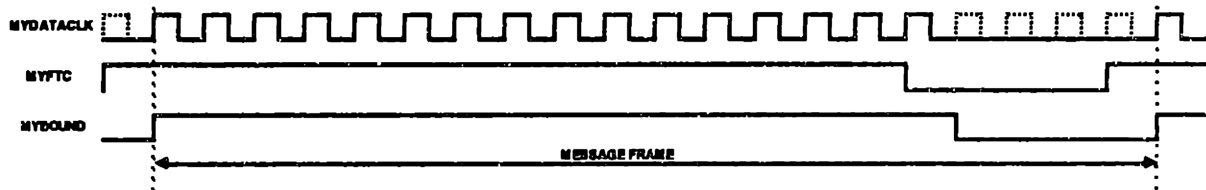


Figure 4.9: Basic Message Transmission Frame Signals of FTC

reliable NE because communication ports are likely sources of failure [Har87]. As a fortunate consequence of the TAXI chip design, it has very low cost in terms of space. As will be shown in the fault tolerant clock discussion, it provides sufficient information to realize a message frame synchronizing circuit. This obviates the need to send separate timing references between FCRs, either as data or as clock signals physically transmitted on other links.

Due to the pipelined architecture of the TAXI chipset, the transmission delay over a minimal length link is approximately 4.5 byte times (562.5ns at 8 MHz).

4.2.4 The NE Fault Tolerant Clock

The NE fault tolerant clock (FTC) is the mechanism which synchronizes the NEs. The synchronization events are message exchanges, thus the FTC design is closely linked to the communication link design, and is fundamental to the operation of the data paths. The basic operation of the FTC circuit is described below.

The FTC mechanism is told by the global controller the length in bytes of the next message (or packet) frame. The FTC generates a signal called MYDATACLK (the local message transmission byte clock) which will contain a number of pulses at the fundamental byte frequency equal to the number of bytes in the packet. The FTC mechanism will then hold MYDATACLK low for a nominal value of 4 byte times before beginning the next frame (Figure 4.9). The pulses of the MYDATACLK waveform are framed by another signal called MYBOUND (Figure 4.9). To synchronize the frame boundaries on the various FCRs, the FTC mechanism compares the MYBOUND signal with the perceived BOUNDs of the other FCRs (taking into account the normal propagation delay of the communication mechanism) and adjusts the number of periods it waits before starting the next frame. This is performed by recovering a BOUND signal from each of the external links, selecting the median BOUND [KSB85] occurrence of all non-faulty external links, and comparing the occurrence of a low to high transition of this median bound with the same transition on MYBOUND. If the local NE perceives itself to be ahead of the others, it will wait five byte times before beginning the next frame. If the local NE perceives itself to be behind the others, it will wait three byte times before beginning the next frame. Otherwise it will wait the nominal four byte times. In this way the BOUND signals will be synchronized among the FCRs.

In addition to creating the synchronized MYBOUND and MYDATACLK sig-

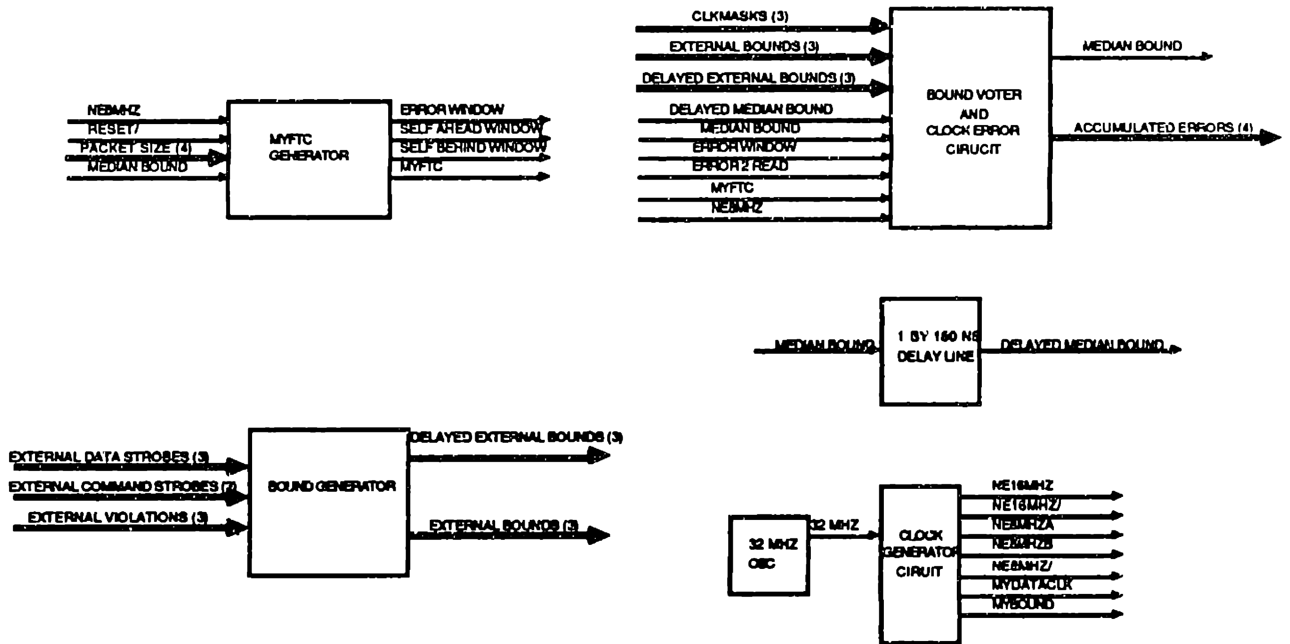


Figure 4.10: FTC Block Diagram

nals, the FTC mechanism is also responsible for generating the various fundamental timing references of the NE. These include both 16 MHz and 8 MHz and their complements. These signals are used to run the global controller and the other various parts of the NE. The 8 MHz signal is related to MYDATACLK in that MYDATACLK is the 8 MHz signal with pulses deleted. However, the 8 MHz and 16 MHZ signals are not phase locked among FCRs. The synchronization between FCRs occurs at the frame level (i.e. the BOUND signals are synchronized). The functional synchronization of the NEs is ensured by providing the global controller with a means of detecting transitions of a signal, MYFTC, which is related to MYBOUND via deterministic translation (i.e. MYBOUND is MYFTC delayed one byte period, Figure 4.9). A high to low transition on MYFTC signals the global controller of the emminent start of the next frame. The controller can then set the correct NE function and packet size for the next frame. The resultant exchange will begin on the rising edge of MYBOUND.

The block diagram of the FTC mechanism consists of four parts (Figure 4.10). They are the following:

1. The clock generator circuit.

2. The MYFTC generator circuit.

3. The bound voter and clock error circuit.

4. The external bound generator circuit.

The clock generator circuit is responsible for creating the fundamental frequencies used in the NE. These are NE16MHZ, NE16MHZ/, NE8MHZ and NE8MHZ/. These are created by dividing the output of a 32 MHz oscillator via a PLD, buffering and inverting the outputs of the divider to achieve good drive capability with minimum skew between the signals. The circuit that does this is shown in the FTC schematic (sheet 4). Part of the clock generator PLD is also used to condense the multiple QERR signals from the voter into a single QERR signal to be used by the syndrome accumulator.

The MYFTC generator circuit is a state machine implemented in a PLD. This machine is responsible for generating the signals of Figure 4.9 based on the packet size information from the global controller. The MYFTC generator must also correctly perform the adjustment of these signals based on the comparison of MYBOUND with the median of the external bounds (MEDIAN BOUND). In addition to generating the signals MYFTC, MYBOUND, and MYDATACLK, this state machine also generates the signals ERRWIN, SAWIN, and SBWIN. These signals frame windows that are used in the MEDIAN BOUND comparison. A rising edge on the MEDIAN BOUND signal while ERRWIN (error window) is asserted denotes that the local NE is in error, either too far ahead or behind of the median. The inactive period of ERRWIN denotes the tolerable skew between the bound signals of the NEs. In the current implementation it is three byte periods. Since the NEs use the bound signals to maintain framewise functional congruence, the occurrence of an identical event (such as delivery of a message to the REC FIFO) on different NEs can be skewed at most three byte times. If the MEDIAN BOUND transition occurs while SAWIN (self ahead window) is asserted, the local NE concludes it is ahead of the others and will wait 5 byte times before beginning the next frame. If the MEDIAN BOUND transition occurs while SBWIN (self behind window) is asserted, the local NE concludes it is behind the others and will wait 3 byte times before beginning the next frame. If neither SAWIN nor SBWIN are asserted when the MEIDAN BOUND transition is detected, the NE concludes it is in synchronization with the others and will wait the nominal 4 byte times before beginning the next frame. The timing of the important signals for these three scenarios are shown in Figure 4.11, Figure 4.12, and Figure 4.13.

The minimum time that the MYBOUND can be low cannot be less than maximum possible skew between bound signals in order that the MEDIAN BOUND detection ciruit work correctly. The positioning of these windows is not without cause. The three period lapse in ERRWIN is centered about the time MEDIAN BOUND would be seen to rise on the local fault containment region if it were identically in phase with MYBOUND (i.e. the offset is the normal inter-FCR communication link propagation delay). This three period region of tolerable inter-FCR skew overlaps the SAWIN and SBWIN each one period. There is also one period in which none of the window signals are asserted. This period is the byte time centered about the expected MEDIAN BOUND time and is called the *self normal* period.

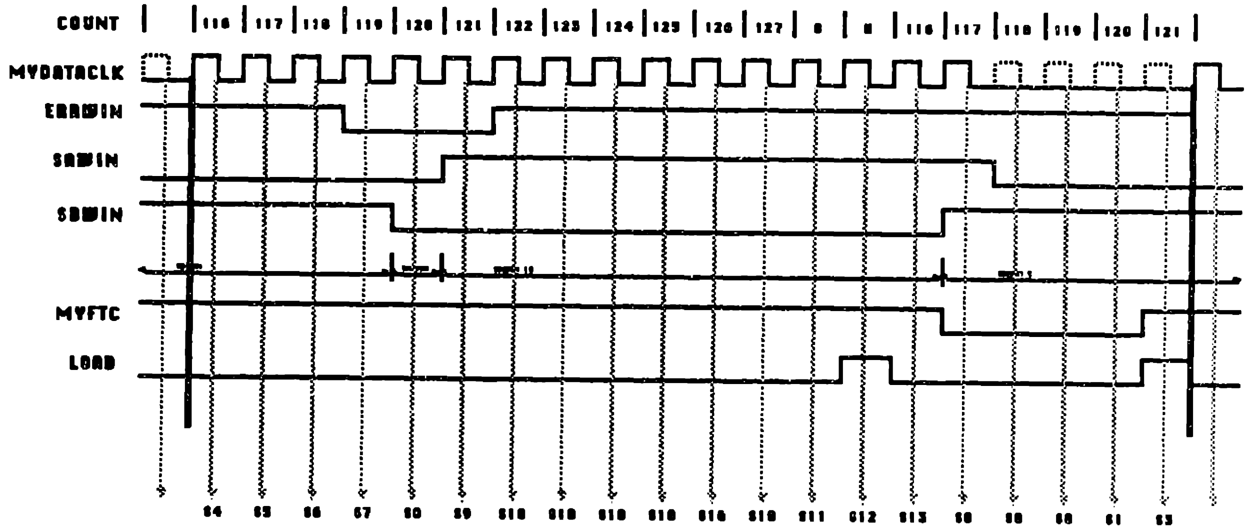


Figure 4.11: Self Normal FTC Timing Signals of FTC

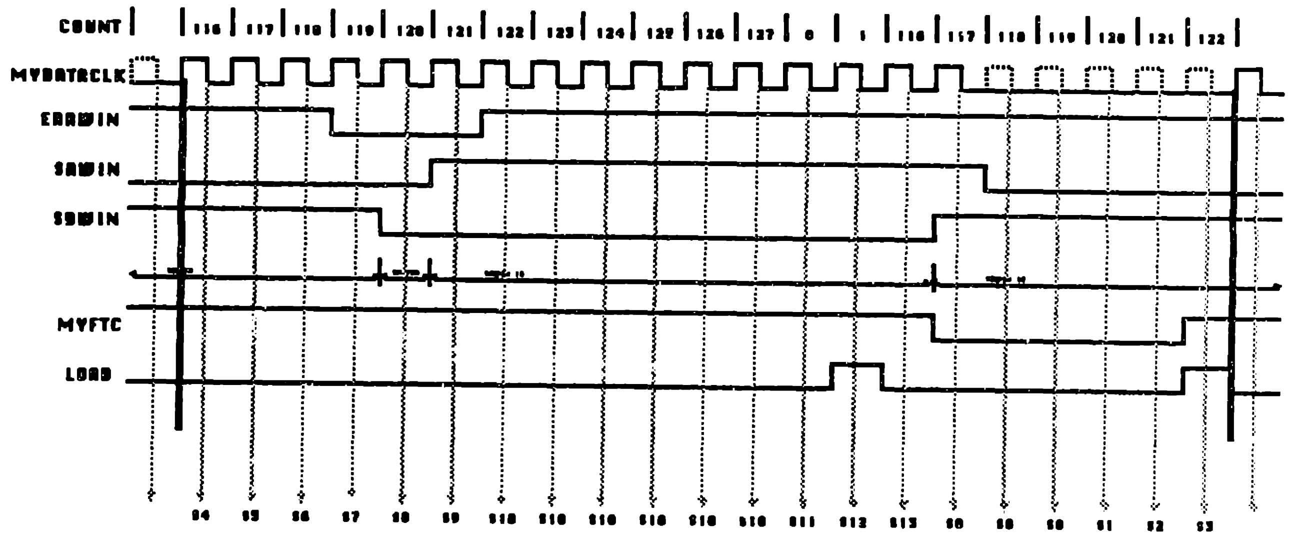


Figure 4.12: Self Ahead FTC Timing Signals of FTC

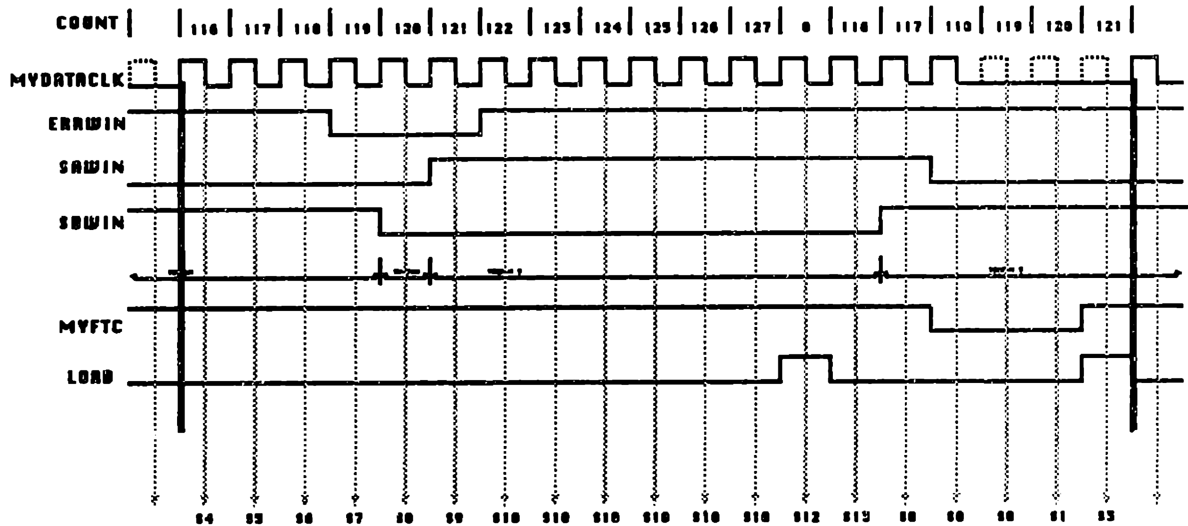


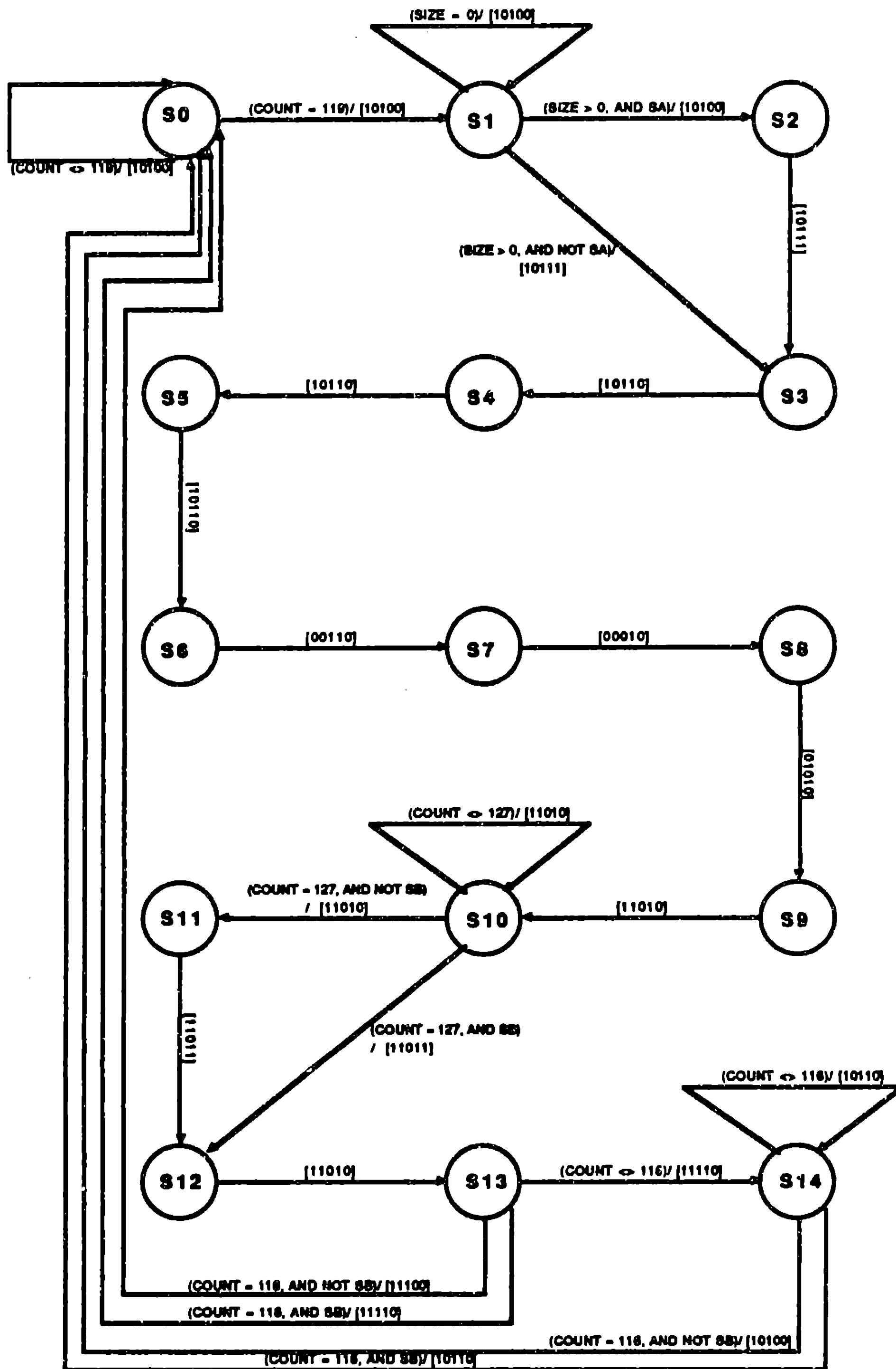
Figure 4.13: Self Behind FTC Timing Signals of FTC

The one period overlap of SAWIN and SBWIN approximately half way through the frame is logically considered part of the SBWIN.

The result is that during steady state operation, a NE will consider itself synchronized with the others if, accounting for propagation delay, its MYBOUND occurs within ±1.5 byte times of the MEDIAN BOUND. A NE will make a nominal 4 period adjustment if, accounting for propagation delay, its MYBOUND occurs within ±0.5 periods of the MEDIAN BOUND. The self ahead and self behind windows are generated to split the entire frame equally such that the adjustment scheme can provide for a convergent initial synchronization. Consider two FCRs that power up with frames arbitrarily out of phase. A given NE will use the other's MYBOUND as its MEDIAN BOUND. In this scenario, positioning of SAWIN and SBWIN is such that the two FCRs cannot repeatedly make the same conclusion about which way to adjust (i.e. they will never both see themselves behind).

To generate the signals shown if Figure 4.11, Figure 4.12, and Figure 4.13, the MYFTC generator is implemented as a state machine in a PLD with an additional seven bit counter. There are 4 state bits, and the state transition diagram (Figure 4.14) illustrates the machine operation. The elementary operation is as follows, the counter is loaded with a count based on the packet size when LD (load) is high. The count is incremented with each 8 MHz clock, and predetermined count values will cause state transition sequences. The other information affecting the state transitions is the result of the MEDIAN BOUND comparison. This information is generated by registering the state of the windows with MEDIAN BOUND. This information consists of two bits, one indicating whether or not the FCR is self ahead (SA), the other indicating whether or not the FCR is self behind (SB). All this occupies a single PLD as shown in the schematic (sheet 4).

The external bound circuit is relatively straightforward. Based on the infor-



State transitions are labeled: (cond)/ [output vector].
Output vector is of the form: [ERRWIN,SAWIN,SBWIN,MYFTC,LD]

Figure 4.14: State Diagram of MYFTC Generator

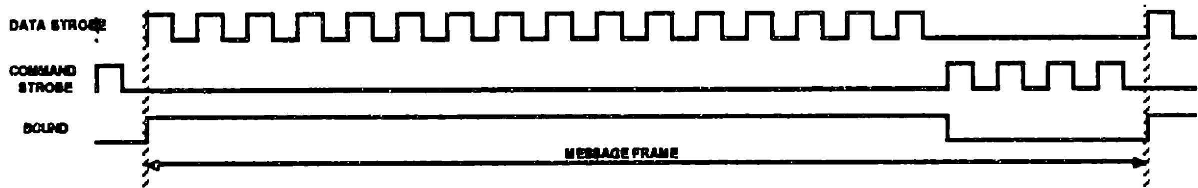


Figure 4.15: Basic Message Transmission Frame; External Signals

mation presented in the inter-FCR communication link section, the typical frame should appear as in Figure 4.15. Therefore, an external bound signal (i.e. LBOUND, RBOUND, or OBOUND) is asserted whenever a data strobe follows a command strobe, and is cleared whenever a command strobe follows a data strobe. The external bound circuit is implemented in a single PLD (sheet 4). In addition to generating the external bounds, this PLD also generates a version of each external bound which is delayed 1.5 periods. These delayed bounds (i.e. DELLBOUND, DELRBOUND, and DELOBOUND) are used in the clock error circuitry.

The final FTC component is the median bound voter and clock error detection circuit. The signal MEDIAN BOUND, which is used to decide the local adjustment, is created by a 3-way maskable voter. The voting function will assert MEDIAN BOUND whenever any two enabled bound signals are asserted. The mask signals (LCLKMASK, RCLKMASK, and OCLKMASK) are generated by the global controller and will be discussed in more detail in that section. The generation of MEDIAN BOUND is required so that the FTC can have a valid reference to adjust to. No single erroneous bound signal can corrupt MEDIAN BOUND [KSB85].

The clock error circuit accumulates clock error information in much the same way as the syndrome accumulator and link error accumulator record syndrome and link errors. The clock error accumulator outputs four bits of the second NE error byte read by ERROR2READ. The MACCERR bit is set whenever the MEDIAN BOUND samples ERRWIN active. The accumulated error bit for a given external clock is generated if any of three scenarios happen.

1. The delayed version of the external bound rises before MEDIAN BOUND is asserted. This corresoponds to the external bound being more than 1.5 periods ahead of MEDIAN BOUND, and is called the *ahead error* condition.

2. DELMEDBOUND, which is MEDIAN BOUND delayed by 1.5 periods, is asserted before the external bound is. This corresponds to the external bound being more than 1.5 periods behind MEDIAN BOUND, and is called the *behind error* condition.

3. The external bound has not experienced a low to high transition since the error accumulator was last read. This corresponds to the external link being failed or not connected, and is called the *presence error* condition.

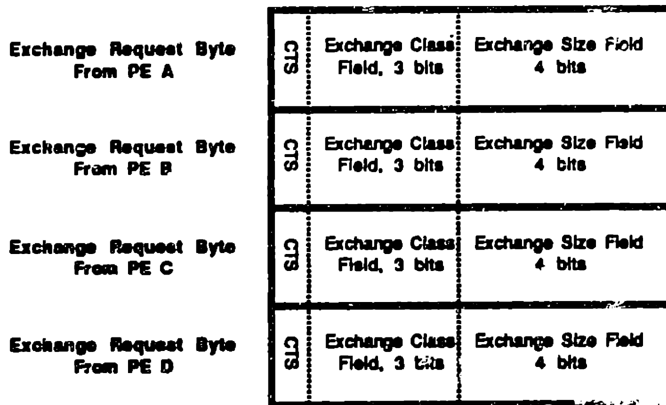


Figure 4.16: SERP Packet Format

Reading the accumulator by asserting ERROR2READ will clear it. The bound voter and error circuit occupy a single PLD (sheet 4) with the exception of DELMEDBOUND generation. Since no timing reference synchronous to MEDIAN BOUND is available, this 1.5 period delay must be done via a tapped delay line.

4.2.5 The NE Scoreboard

The NE scoreboard is the entity which consumes the result of the SERP (System Exchange Request Pattern) exchange, and decides what processor exchange if any can be performed. The scoreboard receives the voted SERP packet which consists of an exchange request pattern byte from each FCR (Figure 4.16).

The exchange request pattern byte consists of 3 items. Three bits are used to specify the exchange type (Figure 4.4). Four bits are used to specify the exchange size. One bit is used to denote the "legitimacy and flow control" of the exchange request byte. When this bit (named CTS for Clear To Send) is asserted the exchange request byte is legitimate (i.e. the class FIFO was not empty when it was read), and the NE flow control situation is satisfied (i.e. the corresponding NE's REC FIFO is not greater than half full, enabling a maximum size message to be exchanged with impunity). Given this information, the scoreboard must perform the two basic functions discussed below.

The scoreboard must vote the exchange class and size fields from each of the four exchange request pattern bytes. This is required to ensure that all NEs achieve consensus concerning what exchange was requested by all non-faulty PEs. The voting that is done by the data path voter merely provides each NE with a consistent SERP packet (Figure 4.16). Recall that in the presence of a single arbitrary failure, the data path voter will insure agreement among healthy NEs of the content of the SERP packet. However, since the SERP packet is constructed of 4 input consistency exchanges, the validity (read correctness) of any single exchange request pattern byte is not required. Therefore an additional vote of the individual exchange request

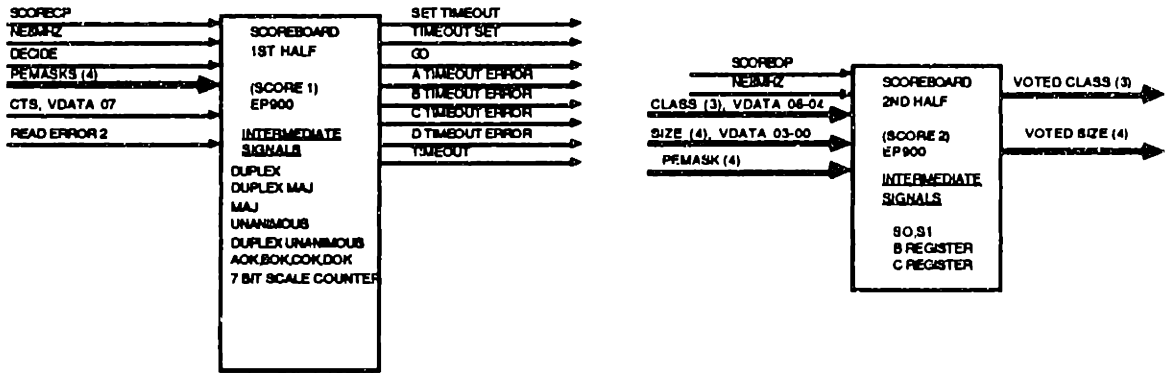


Figure 4.17: Scoreboard Block Diagram

pattern bytes is required to produce a valid exchange class and size. This scoreboard voting is performed by the "second half" of the scoreboard (Figure 4.17). The voting is performed by a byte wide, maskable, serial, bit-for-bit majority voter[4] implemented in a PLD (sheet 8). It is a serial voter in that it votes four bytes that arrive at the byte wide input at successive byte times. The exchange request pattern bytes are gated off the VOTED DATA bus via the SCOREOP control signal. This voter temporarily stores the four bytes as they arrive and upon receiving the last, votes them according to the same function used in the data path voter. The masks used in this process are the PEMASKs supplied by the global controller. More will be said about these masks when discussing the global controller. It is interesting to note that the scoreboard deals in virtual FCR identifiers. This is because the exchange request bytes that are input to the scoreboard are identified in terms of virtual identifiers. The *scoreboard voter* does not provide any syndrome information due to lack of space in the PLD.

The second basic scoreboard function is to decide when all non-faulty PEs have submitted an exchange request pattern byte. This is performed by the scoreboard "first half" (Figure 4.17) by anlayzing the CTS bit. An exchange request pattern is assumed to be legitimate when the CTS is set. Legitimacy simply implies that it was not the result of reading an empty class FIFO, and that sufficient room exists in the REC FIFO to accept the result of the requested exchange. PEs are said to have submitted an exchange request pattern byte when their CTS bits are set. A PE is faulty, by definition, if its corresponding CTS bit is not set within an a *priori* determined time after the majority of CTS bits are set. This time must be at least as long as the maximum allowable processor skew. This time is measured in terms of a number of SERP exchanges as a count called the TIMEOUT, and is programmed via the *timeout counter* and the *scale counter*. If the CTS bits are unanimously set at a given SERP exchange, the corresponding exchange output

[4]This *scoreboard voter* is identical to that designed and implemented by Stuart J. Adams for the CSDL FTPP.

by the scoreboard voter will be performed during the next frame by all NEs. If a majority of CTS bits are set the timeout process will be started. The message will be enabled following the SERP exchange that causes either a unanimous assertion of CTS bits, or the expiration of the timeout. The definitions of majority and unanimous depend on the current configuration of the PEs (i.e. how many healthy PEs are in operation). This information is specified via the PEMASK from the global controller. When a message is enabled, the scoreboard records four bits of error information denoting whether a given PE requested the exchange. These four bits are called *timeout error* and comprise the remainder of the second NE error byte that is read with ERROR2READ. A given PE's timeout error flag is set if its CTS was not set at the time the message was enabled. Operation of the scoreboard "first half" is controlled by the signals SCOREOP and DECIDE. Data is gated into this half of the scoreboard via SCOREOP (sheet 8). The four CTS bits are stored as they are input, and then compared. The global controller asserts the DECIDE signal at the end of the SERP exchange. The DECIDE signal causes the scoreboard to do one of the following:

1. Assert GO if the CTS bits are unanimously set. This informs the global controller to execute the exchange as directed by the output of the scoreboard voter.

2. Assert SET TO (set timeout) if a majority, but not all, of CTS bits are set, and the timeout is not already set (TO SET = 0).

3. Assert TO SET if not already set and SET TO = 1. This will clear SET TO.

4. Assert GO if TO SET is set and TIMEOUT occurs. This informs the global controller to execute the exchange as directed by the outputs of the scoreboard voter. TIMEOUT is asserted when the scale counter and timeout counter (which are clocked with DECIDE and thus count SERP exchanges) reach a programmable value. Currently the minimum value is 2 and the maximum value is 16,384. Given that a SERP exchange takes 58 byte times, this corresponds to a maximum allowable processor skew of 0.12 seconds and a minimum of $14\mu s$.

Asserting DECIDE always causes the scoreboard to latch any timeout errors. A timeout error is asserted if GO is asserted and the corresponding PE's CTS bit is not.

A typical scenario is illustrated below (Figure 4.18). The time line shows processor requests for a *From Vote* exchange of 16 bytes arriving at the NE aggregate. In this scenario PE B writes an invalid request byte.

At time t_{serp1} the NEs read their class FIFOs and begin a SERP exchange. The voted SERP packet sent to the scoreboard will have none of the CTS bits set, hence at $t_{decide1}$ the NEs will decide that no exchange request can be performed.

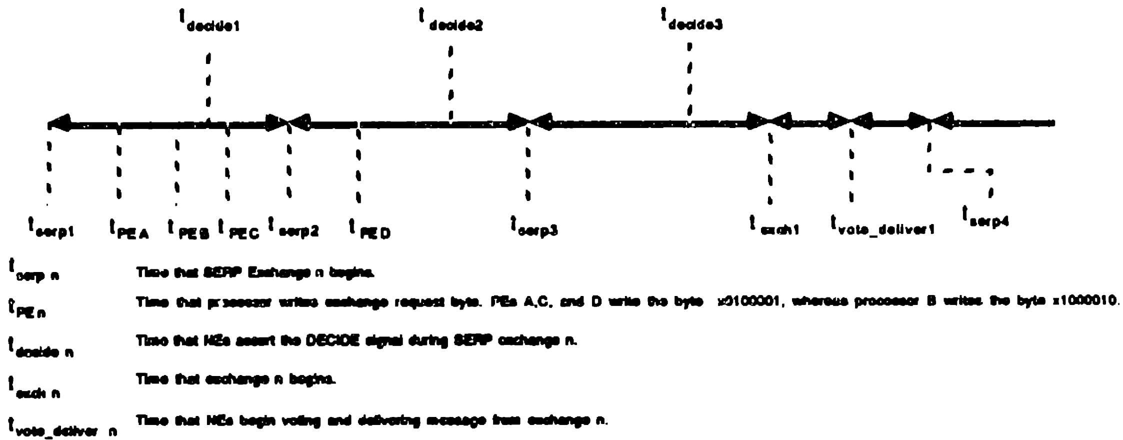


Figure 4.18: NE Operation Time Line

	CTS	Exchange Class Field, 3 bits	Exchange Size Field 4 bits
Exchange Request Byte From PE A	1	010	0001
Exchange Request Byte From PE B	1	100	0010
Exchange Request Byte From PE C	1	010	0001
Exchange Request Byte From PE D	0	XXX	XXXX

Figure 4.19: SERP Packet Delivered to Scoreboard: SERP 2

Another SERP exchange will begin with the NEs reading their class FIFOs at time t_{serp2}. The voted SERP packet will appear as shown in Figure 4.19. A majority of processors have their corresponding CTS bits set. Therefore at time $t_{decide2}$ the NEs will begin the timeout process by asserting SET TO.

Another SERP exchange will ensue. At time t_{serp3} the NEs will read their class FIFOs. The resultant voted SERP packet will appear as shown in Figure 4.20. All the CTS bits are set, hence at time $t_{decide3}$ the NEs will assert GO indicating that a PE exchange can be honored.

The exchange class and size fields are voted across request bytes resulting in a correct values. Scoreboard voter errors are not recorded. The *From Vote* exchange will take place at time t_{exch1}. Another FTC cycle is required to vote and deliver this message to the REC FIFO. SERP cycles resume at time t_{serp4}.

	CTS	Exchange Class Field, 3 bits	Exchange Size Field 4 bits
Exchange Request Byte From PE A	1	010	0001
Exchange Request Byte From PE B	1	100	0010
Exchange Request Byte From PE C	1	010	0001
Exchange Request Byte From PE D	1	010	0001

Figure 4.20: SERP Packet Delivered to Scoreboard: SERP 3

4.2.6 The NE Global Controller

The Global controller is the central micro-controller for the NE. The controller consists of many parts (Figure 4.21) which will be introduced here, and discussed in further detail below. The heart of the controller is a finite state machine implemented by five, 2k by 8, registered, programmable, read only memories (RPROMs)[5]. The RPROM bank has 40 outputs and 11 inputs and is clocked at 16 MHz (twice the byte frequency). Ten of the outputs form the CTL (control) bus which is used to provide the state of the machine via the *stack pass thru register*. The stack pass through register allows these ten bits to be passed back to the PROM inputs to form the next state address. The stack pass thru register also provides for an auto-incrementing, one deep subroutine register. This allows the CTL BUS lines to be used as control outputs. The eleventh input to the state machine is driven by the *conditional multiplexor* (MUX). The MUX allows important signals to be sampled by the controller to specify control branches. Four RPROM outputs (called SELECTs) are used to select the desired conditional bit. These SELECTs are also used to control an *output decoder*. The output decoder enables the machine to assert even more control outputs. The output decoder is clocked by the inverse of the signal which clocks the RPROM bank (NE16MHZ/). Therefore, the output decoder is useful for control actions which require RPROM outputs to have stabilized. The global controller also has access to a port of the dual-port RAM. This is provided by a more generalized 8 bit data-port access to the VOTED DATA bus. The global controller's port of the dual-port RAM is accessed via the VOTED DATA bus. The global controller can access the VOTED DATA bus through a register connected to the CTL bus. The address for the NE side of the dual-port is driven by the *external address (EXADDR) latch* which is also resident on the CTL bus. There

[5]The design of the basic controller state machine is copied from an identical controller built for the CSDL FTPP by Stuart J. Adams.

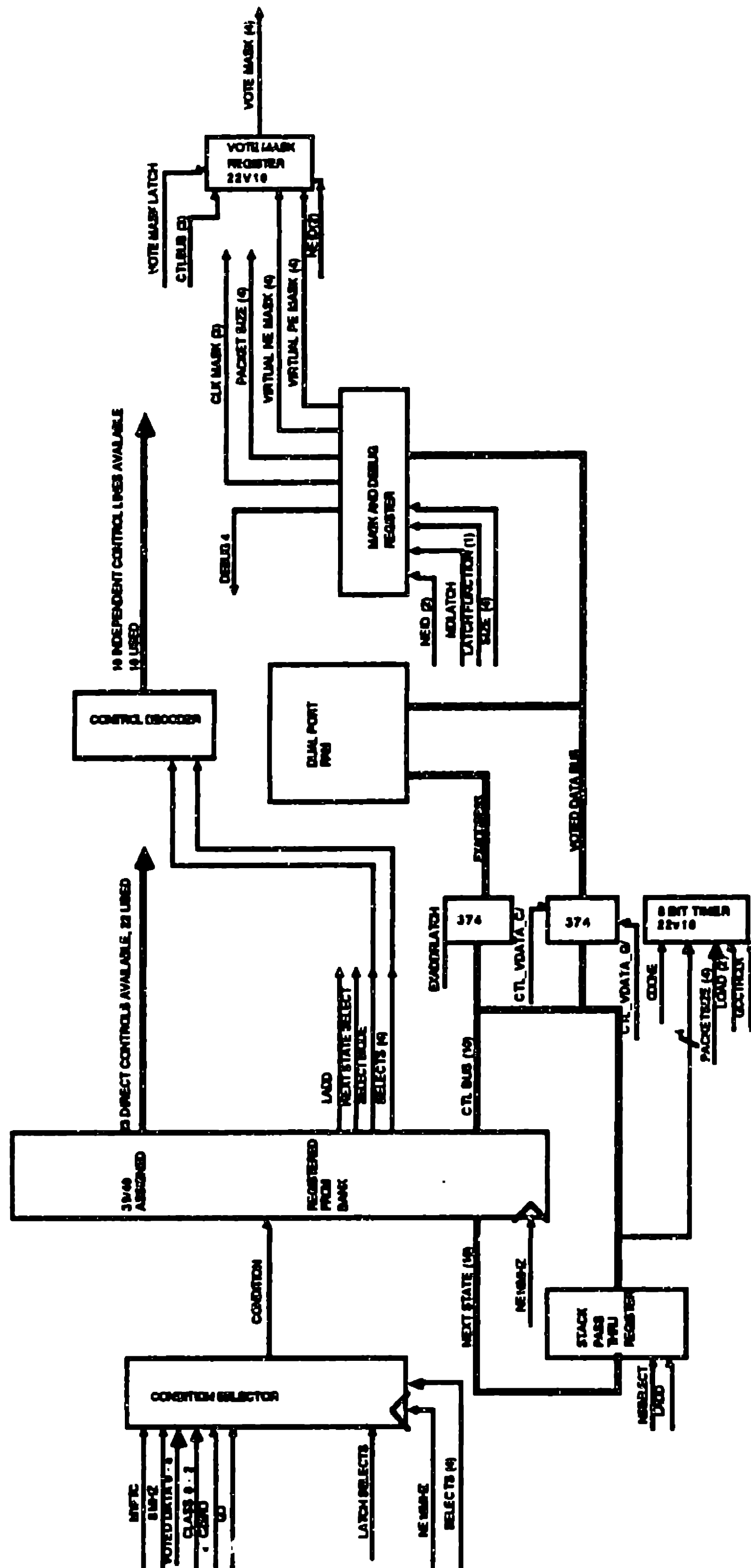


Figure 4.21: NE Global Controller Block Diagram

is another important controller entity resident on the VOTED DATA bus. This is the *mask, size, and debug register.* In debug mode, this register allows the global controller to set various parameters of the NE operation. These include the masks, the packet (frame) size, the clear selects, and the data path debug enable signals. In operational mode, this register is where the voted value of the masks will reside. Finally, the controller also has access to a counter via the CTL bus. This eight bit counter can be loaded in several modes. One important mode allows the counter to be loaded with the packet size output by the scoreboard voter.

The stack pass thru register consists of two pieces (sheet 9). The first piece is a latch which allows the next state of the controller to be determined by the value of the CTL bus. The other piece is a PLD which implements the subroutine register function of the stack pass thru register. The operation of the ensemble is controlled by two RPROM outputs NSSELECT (Next State SELECT) and LADD (Latch ADDress). Setting NSSELECT = 0 enables the ensemble in pass through mode. To use the CTL bus as control outputs a subroutine must be entered. Asserting LADD in state N will cause the value of the CTL bus + 1 to be stored in the subroutine register. If the value of the CTL bus was N+1 in state N, the address N+2 is latched at the subroutine register. The CTL bus still drives the next state if NSSELECT = 0, hence the machine will move to state N+1. In state N+1, the CTL bus lines may be used as outputs, and the next address can be specified as N+2 by asserting NSSELECT. The subroutine register has an auto-incrementing capability. Asserting LADD and NSSELECT together enables this mode. In this mode the contents of the subroutine register will be automatically incremented and used as the next state of the machine. This provides a means of implementing extremely long sequential subroutines.

The conditional multiplexor also consists of two pieces (sheet 9). The first piece is a straight forward registered multiplexor. Four RPROM outputs are used as SELECTs, therefore 16 inputs are possible. Currently only six select modes are used. The top half of the MUX handles four of these. They include the following:

1. COUNTDONE. This signals the counter value is zero.

2. MYFTC. This is generated by the FTC circuit and is used to synchronize the controller to the frame boundaries.

3. DELNE8MHZ. This is a slightly delayed version of the byte clock. It is provided to enable the controller to sync to the rising edge of the byte clock.

4. GO. This is the scoreboard output which signals that a valid processor request is ready to be exchanged.

Any of these inputs can be selected by the appropriate SELECT value and assertion of the signal SELECTMODE. SELECTMODE signals the MUX that the present state of the SELECT lines represents a new condition request. The value of SELECTs will be saved as LATCHEDSELECTs. This enables the same SELECTs to

be used to drive the output decoder without changing the conditional input selected. A conditional input is thus selected when the SELECTs are set to the corresponding value and SELECTMODE is asserted, or when LATCHEDSELECTs are set to the corresponding value. The MUX is registered, hence if selects are asserted in state N, the branch cannot occur until state N+2.

The second half of the MUX is used to select a sequence of a group of bits. Two such groups are used. The first group consists of the low order 5 bits of the VOTED DATA bus. These bits correspond to the debug command value when the controller reads the debug command register. They also correspond to the voter syndrome. A SELECT value of 14 selects this group of bits. The second half of the MUX then sequences through these bits repeatedly until a new SELECT value is latched at the MUX with SELECTMODE. The sequencing is done with the help of an internal three bit counter. In this way the global controller can parse the debug command using the single condition input. Additionaly, the controller can sample the voter syndrome, during the initial synchronization routine, to determine when there is no voter errors. Using the MUX in this manner incurs an additional pipeline delay. Therefore if the SELECTs are asserted in state N, the first bit is available to branch on at the controller in state N+3. The other group of bits provided by this part of the MUX is the voted exchange class outputs of the scoreboard voter. These are used by the controller to identify the exchange type to be performed following assertion of GO.

The controller output decoder is implemented in a PLD (sheet 9). Output selection is based on the value of the SELECTs when SELECTMODE = 0. Though 15 such outputs would be usable, only 10 will fit in the PLD. The primary motivation for the decoder is to make use of the SELECT outputs when not using them to specify conditional inputs. The control signals that are commanded by the decoder cannot, of course, be asserted simultaneously. Not much luck was had finding obvious candidates for the decoder. The reset signals for the various FIFO memories of the data paths and the VME subsection were stuck there, though this proved to be cumbersome later. One good match was found however. This was in the group of signals that dealt with the CTL bus interface to the dual-port RAM and the mask, size and debug register. In writing the code to access these appendages of the controller it was useful to have the half a period for CTL bus signals to stabilize before latching them at the VOTED DATA interface registers. This allowed for such accesses to take only a single state instead of the two or three that might have been required otherwise.

The global controller has an eight bit counter implemented in a PLD (sheet 10). The counter clock strobe is a controller decoder output, and a COUNTDONE flag signaling a count value of zero is a possible conditional input to the controller. This makes it a fairly generic event counter. The counter has four modes specified by the two GLOBALCTRLD bits. A clock pulse while the load mode is 0 causes the counter to be decremented. Clocking the counter with a load mode of 1 causes the

PACKETSIZE output of the scoreboard voter to be multiplied by 16 and loaded. Clocking the counter with a load mode of 2 causes the lower four bits of the counter to be loaded with the lower four bits of the CTL bus. The upper four count bits remain unaffected. Clocking the counter with a load mode of 3 causes the upper four bits of the counter to be loaded with the lower four bits of the CTL bus. The lower four count bits remain unaffected. The counter was implemented in this fashion to remove the need for the controller to parse the exchange size information as conditional input as it does the exchange class information.

The global controller's port of the dual-port RAM will be discussed in conjunction with the mask, size and debug register and the vote mask register. The controller specifies the address to be used in a dual-port access by writing the value to the EXADDR latch on the CTL bus (U2777, sheet 10). The controller can write data to the dual-port over the VOTED DATA bus via the CTL bus (U2781, sheet 10). The controller only reads data from the dual-port in two scenarios. The first is when it reads the debug command register. To do this, the controller commands the dual-port to assert the value of the command register location onto the VOTED DATA bus. The controller can then "read" these bits via the conditional MUX. The second read scenario is when the controller changes some parameter stored in the mask, size and debug register (sheet 10). To do this, the controller commands the dual-port to assert the value of a given location onto the VOTED DATA bus, and then latches this value at the mask, size and debug register.

The mask, size and debug register is a hodgepodge of logic which performs two principle functions. The first is to specify the current configuration of the NEFTP as reflected in the state of the various masks signals. To begin with, lets discuss the normal operation of this register and the masks. The masks refer to the four PEMASK signals and the four NEMASK signals that make up the NEFTP mask byte. The mask byte uses the virtual FCR identifiers and represents the configuration of the NEFTP. A mask bit value of 1 indicates that the corresponding entity is operative, while a 0 indicates it is failed and is "masked out". As an aside, the PE is a subset of the NE in terms of fault containment. A NE whose PE has failed is still of use to the aggregate system while the reverse is not. Hence a mask byte which masks out the NE and not the PE is not logical. This situation is prohibited by the mask, size and debug register. The mask byte is written to this register by use of the *FromV with masks* exchange. This exchange type specifies that the first byte of the message will be the new mask byte. The value of the mask byte, the result of an output consensus exchange, is latched into the mask size and debug register as it comes out of the voter. The value of the NEMASKs are as specified on the lower half of the VOTED DATA bus, the value of the PEMASKs are the logical AND of the two halfs of the VOTED DATA bus. From the NEFTP mask byte, the mask, size and debug register, in conjunction with the voter mask register, generates all the masks to be used in the NE. These include:

- the data path voter masks, which are physical masks,

- the FTC masks, which are physical masks,
- the scoreboard masks, which are virtual masks.

The masks used by the data paths voter depends on the exchange type. During an output consistency exchange the VOTEMASKs are the logical AND of the PEMASKs and the NEMASKs translated to physical terms by the NEID. During a two round input consistency exchange, the VOTEMASKs are the NEMASKs translated to physical terms by NEID minus the source channel. For example, when voting the result of a *FromA*, the source channel (A) is masked out. The VOTEMASKs are generated by the vote mask register which implements the above function using the NEMASKs and PEMASKs from the mask, size and debug register, and the NEID.

The masks used in the FTC are generated in the mask, size and debug register. These three CLKMASKs are simply NEMASKs translated by the NEID less the local FCR. This translation is performed in the mask, size and debug register.

The masks used in the scoreboard are exactly the PEMASKs as latched in the mask, size and debug register.

The second principle function of the mask, size and debug register is to register the packet size information for the FTC mechanism. Therefore, the size information output of the scoreboard (VSIZE) is input to the mask, size and debug register. VSIZE can be registered to PACKETSIZE which is output from the mask, size and debug register to the FTC mechanism, and the controller counter (sheet 10).

The mask, size and debug register has some secondary functions.

- It provides for the specification of the mask and size information in debug mode. In debug mode, the source of this information can be either the dual-port RAM, or the global controller. In this way, the mask and size information can be user specified via the Interactive Debugger, or it can be coded into the global controller debug routines.

- It provides for the specification of the DBEN signals which control the operation of the debug router. Again, these signals can be specified by the user via the Interactive Debugger and the dual-port, or they can be specified by the global controller.

- It provides the virtual to physical translation mechanism for the data path FIFO clear select signals. To clear a specified data path FIFO, the controller writes two bits of information to the mask, size and debug register which are a virtually encoded identifier of the FIFO (e.g. "clear the data path FIFO corresponding to FCR A"). This value is translated by the NEID and stored in physical form as the CLEARDATAPSEL signals. The CLEARDATAPSEL signals tell the asynchronous controller what physical FIFO (i.e. the FIFO corresponding to the LEFT FCR) to clear when the CLEARDATAP signal is asserted. This was done so that the microcode for the NE global controller could be FCR-independent.

LATCH FUNCTION	CTL DATA 7 - 5	FUNCTION
0	X	Latch virtual PE and NE masks. Operational mode.
1	001	Latch clock mask. Debug mode.
1	010	Latch size. Operational mode.
1	011	Latch size. Debug mode.
1	100	Latch DBENs. Debug mode.
1	101	Latch Data Path FIFO Clear Selects. Either mode.

Figure 4.22: Mask, Size and Debug Register Latch Functions

The mask, size and debug register is shown in sheet 10, and a table reviewing its funcitons is shown in Figure 4.22. A listing of all the global controller control outputs and their functions is presented in Figure 4.23.

4.3 Status of The NE Prototype

The design presented in this chapter has been implemented in a wirewrap prototype version. The entire NE fits on a single 6U by 160 VME wirewrap panel (see board layout Appendix A, sheet 13). Four such NEs have been built and all phases of operation tested in stand alone mode. These include:

1. debug mode,
2. initial synchronization mode,
3. and operational mode.

A NEFTP has been constructed using these NEs and commercially available 32 bit single board computers, and is currently undergoing testing. The NEFTP is housed in a single 19" VME subrack. The subrack is physically divided into four FCRs in that there exists four independent backplanes. However, all FCRs are powered with a single switching power supply for laboratory demonstration. The current NEFTP provides room in each FCR for an additional wirewrap card or an independent power supply.

CONTROL OUTPUT NAME (EPROM OUTPUTS)	CONTROL OUTPUT FUNCTION (EPROM OUTPUTS)
DPRAMCE/	Chip enable for NE access of Dual Port.
DPRAMRW/	Read, write control for NE access of Dual Port.
DPRAMOE/	Output enable for NE access of Dual Port.
CTL_VDATA_G/	Enable for VOTED DATA / CTL bus interface register.
VMEDBEN/	Enable for VME INTERFACE subsection debug wrap register.
GLOBALCTRLD1	First load function select line for global controller counter.
GLOBALCTRLD0	Second load function select line for global controller counter.
MDLATCHFUN	Latch function bit for Mask, Size and Debug register.
XMITFIFORS	Read strobe for XMIT FIFO.
RECFIFOWS	Write strobe for REC FIFO.
CLASSFIFOOE/	Output enable for class FIFO.
CLASSFIFOSO	Shift out signal for class FIFO.
SCOREOP	Scoreboard control signal.
ERROR1READ/	Read strobe for first of two NE error bytes.
ERROR2READ/	Read strobe for second of two NE error bytes.
REFENABLE	Data path control signal which enables reflected messages to the MY EXTERNAL DATA bus.
REFSEL1	First of two select bits specifying the virtual ID of the data path FIFO to reflect from.
REFSEL0	Second of two select bits specifying the virtual ID of the data path FIFO to reflect from.
NEFUNCTION1	First of the two NE function select bits.
NEFUNCTION0	Second of the two NE function select bits.
VOTINGSERP	Signal identifying that a SERP packet is being voted. Used by SYNDROME ACCUMULATOR to suspend accumulation.
VOTEIN	Signal controlling data path voter operation.
LADD	Latch address at Stack Pass Thru Register.
NSSELECT	Select for Stack Pass Thru Register.
SELECTMODE	Select whether Mux or Output Decoder interprets SELECTs.
SELECT3	Select signal for Mux or Output Controller.
SELECT2	Select signal for Mux or Output Controller.
SELECT1	Select signal for Mux or Output Controller.
SELECT0	Select signal for Mux or Output Controller.
CONTROL OUTPUT NAME (DECODER OUTPUTS)	**CONTROL OUTPUT FUNCTION (DECODER OUTPUTS)**
GLOBALCTRCLK	Clock for the global controller counter.
CLEARDATAP	Data path FIFO clear strobe.
RESETXMIT/	XMIT FIFO clear signal.
RESETREC/	REC FIFO clear signal.
DECIDE	Strobe which terminates SERP exchange. Used to latch results at Scoreboard
KICKDOG/	Strobe to restart Watch Dog timer.
CTL_VDATA_C	Clock for the register which interfaces the VOTED DATA and CTL buses.
MDLATCH	Latch strobe for the Mask, Size and Debug register.
EXADDRLATCH	Latch for the External Address register.
VMASKLATCH	Latch for the Vote Mask register.

Figure 4.23: Global Controller Control Outputs

Chapter 5
Evaluation of The NEFTP

5.1 Reliability Analysis

This section investigates the reliability of the NEFTP architecture and compares it with that of a conventional simplex processor. The approach is to use a continuous time, discrete state Markov model to calculate the state occupancy probabilities of the two systems. Several probabilities pertinent to reliability are recorded. The main probability of interest is the *probability of system loss*, $p_{sysloss}$, at time t given that the system was operational at time $t = 0$. Reliability can thus be defined as $1 - p_{sysloss}$. The probability of system loss is the sum of the state occupancy probabilities (p_i) of all states i where the system is assumed to have failed. The system is considered to have failed when it suffers an uncovered failure of any of its components. The probability of system loss will also include the probabilities (p_i) of all states where the system is assumed to have been shutdown if such shutdown is tantamount to failure. In systems employing redundancy, system shutdown is a function of the redundancy management scheme. The two state probabilities of interest are denoted p_{unsafe} and $p_{shutdown}$; their relationship to $p_{sysloss}$ is dependent upon the application. For example, consider an application such as a nuclear reactor power plant control. High reliability is required because the cost of unsafe failure is horrific. A controlled shutdown would not have the same consequences due to the existence of backup methods of power generation. For such an application, $p_{sysloss}$ would be equal to p_{unsafe}. However, in applications such as autonomous space vehicle control, the shutdown of the control computer may likely be just as catastrophic as an unsafe failure. This is because the shutdown of the control of an autonomous spacecraft may make it uncontrollable trash floating in space, no less a catastrophic event than an unsafe control failure, which rockets the vehicle out of the solar system. For this application, $p_{sysloss}$ would be equal to the sum of p_{unsafe} and $p_{shutdown}$. Applications of the former type will be called critical, and applications of the latter type will be called ultra-critical. Systems employing redundancy have some flexibility in how to manage that redundancy to minimize $p_{sysloss}$. For such systems, the choice of redundancy scheme will be influenced by the criticality of the application.

5.1.1 Definitions and Assumptions

Several assumptions have been made in formulating the Markov models used in the analysis. These assumptions, along with corresponding terminology, are discussed below.

Component failures, such as processor or network element failures are assumed to be independent, and to occur at a fixed rate (λ_{pe} and λ_{ne}, respectively), expressed in failures per hour. Numerical values for λ_{pe} have previously been estimated extensively. A typical value for λ_{pe} is 10^{-4} [Wen78]. A value for λ_{ne} is not readily obtainable. However, the NE implementation is significantly less complex than that of the PE, measured in terms of number of transistors. Hence, a realistic λ_{ne} is arguably less than λ_{pe}. Reliability analysis of a more complex NE architecture intended for hosting multiple PEs [Har87] estimated λ_{ne} at $1.4 * 10^{-5}$. This value should be an overestimate of the λ_{ne} for the NEFTP. As this will make for conservative results, the analysis of the NEFTP uses this value of λ_{ne}.

Reconfigurations are assumed to take place in both the simplex processor and the NEFTP. Reconfiguration in the simplex processor is assumed to consist of some rollback or retry strategy. In the NEFTP architecture, reconfiguration involves isolating and masking out the failed component in the case of a permanent or intermittent fault; or realigning and resynchronizing a component in the case of a transient error. The rate at which reconfigurations can take place may be different when reconfiguring around a failed processor than when reconfiguring around a failed network element. These two reconfiguration rates, expressed in terms of reconfigurations per hour, are μ_{pe} and μ_{ne}, respectively. The reconfiguration time $1/\mu$ includes the fault latency time [SL84]. For simplicity μ_{pe} and μ_{ne} are assumed to be identical and equal to 10^3 per hour in the subsequent analysis [HSL78]. Empirically determined values of μ_{pe} and μ_{ne} would yield more accurate results; however, such empirical analysis must await the final definition and implementation of the fault detection, isolation and recovery (FDIR) software.

All temporal categories of faults are intended to be included in the above mentioned component failure rates. Again, these categories are transient, intermittent, and permanent. In the analysis below, the occurrence of intermittent and permanent faults will result in the same state transitions. This is a result of the mechanism used to detect and isolate faults. In most fault tolerant computers, failure diagnostic information is continually recorded. Failures that are seen to occur a sufficiently large number of times are denoted permanent, otherwise they are transient. Thus the only distinction that need be made in the following analysis is that of permanent *vs* transient failures. Earlier studies [SS82] have suggested that transient faults comprise 80 to 90 percent of all hardware failures. The models used below incorporate this fact in the parameter f_t, the probability that a failure is transient. The default value, $f_t = 0.5$, will yield quite conservative results. As the analysis will show, a smaller f_t results in a larger $p_{sysloss}$; permanent failures are more detrimental to system reliability than transient ones.

All models used below incorporate the parameter c which denotes the probability that a given fault is covered[1] by the system. A simplex processor can achieve a certain degree of fault coverage through use of self tests, timeouts, retry and rollback

[1]See definition of coverage in chapter 2.

techniques and others. The corresponding value of c is realistically between 0.8 and 0.95 [SS82]. The NEFTP is a 1-Byzantine resilient architecture and therefore has unity coverage of a single failure. It may be desired to continue operating the NEFTP after the occurrence of the first permanent failure, although with a reduced reliability, rather than shut it down. In such a case, the model must be adjusted to account for non-unity coverage of subsequent faults.

5.1.2 Simplex System Reliability Analysis

The Markov model of a conventional simplex processor is shown in Figure 5.1. The first state is the *zero failure state*. The occurrence of a covered processor failure moves the system to state 2 which is the *retry/rollback state*. Successful retry is accomplished if the fault is transient. Thus, the system moves back to state 1 with the probability $\lambda_{pe} * f_t$. However, a permanent failure results in an unsuccessful retry. Therefore, the system moves from state 2 to state 3 with a probability of $\mu_{pe} * (1 - f_t)$. In state 3 the system knows that it is faulty, though the fault has been covered, and decides to shut itself down. State 3 is therefore called the *shutdown state*. Finally, from state 1, the occurrence of an uncovered processor failure, transient or permanent, moves the processor to state 4. In state 4 a failure has gone uncovered, which is equated with system failure. State 4 is thus the *unsafe system failure state*. Figure 5.2 plots p_{unsafe}, $p_{shutdown}$, and their sum for the simplex model. In this plot and following plots of probabilities, the default parameters have been used unless otherwise mentioned, and the time axis is in units of hours.

Figure 5.2 illustrates two very important points. First, the suggestion in Chapter 2 that a simplex processor is inadequate for high reliability applications is affirmed. The probability of system loss at the end of a 10 hour mission for an ultra-critical application is approximately $0.6 * 10^{-3}$. The probability of system loss at the end of a 10 hour mission for a critical application is approximately $0.2 * 10^{-3}$. In either case this is clearly unsatisfactory for the applications presented in Chapter 2 which require values of $p_{sysloss}$ on the order of 10^{-7} to 10^{-9} for a 10 hour mission.

These results illustrate why current efforts in fault tolerant computing employ harware redundancy. The probabilities cited above "improve" (i.e. are reduced) as the parameters c and f_t are increased. Figure 5.3 illustrates how p_{unsafe} varies with c. The ordinate of Figure 5.3 is a linear scale to illustrate the affect of c. Note that c merely affects the ratio of $p_{shutdown}$ to p_{unsafe}; c cannot improve their sum. Figure 5.4 demonstrates how $p_{sysloss}$ varies with f_t. Note that f_t does not affect the ratio of $p_{shutdown}$ to p_{unsafe}: f_t does however affect $p_{sysloss}$. A larger value of f_t means fewer permanent failures, and therefore a lower value of $p_{sysloss}$. However such improvements are insufficient to attain the necessary reliability levels.

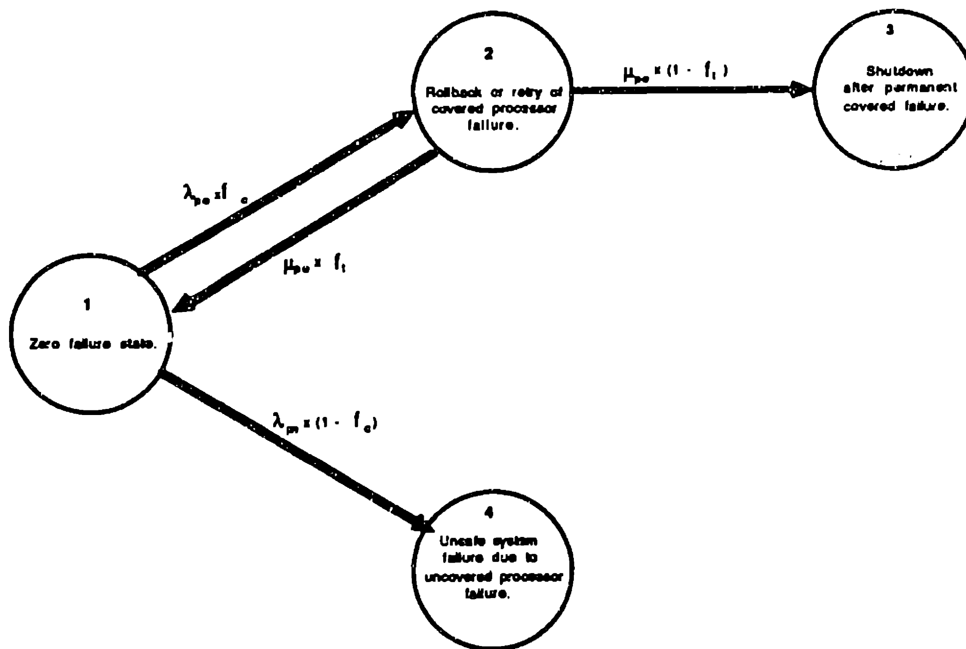


Figure 5.1: Simplex Processor Markov Model

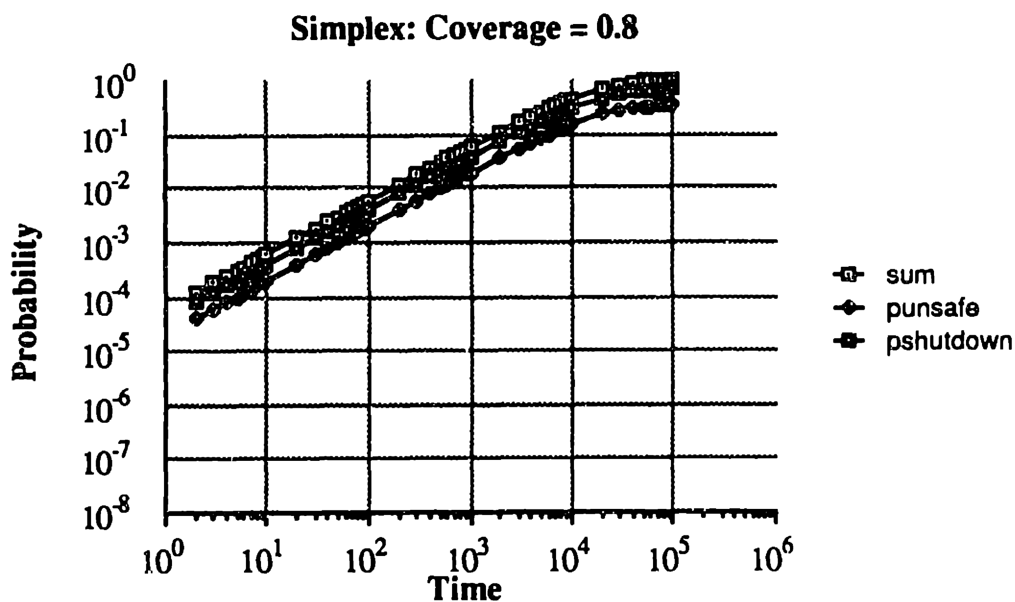


Figure 5.2: $p_{unsafe}, p_{shutdown}$, and $p_{unsafe} + p_{shutdown}$ For A Simplex Processor

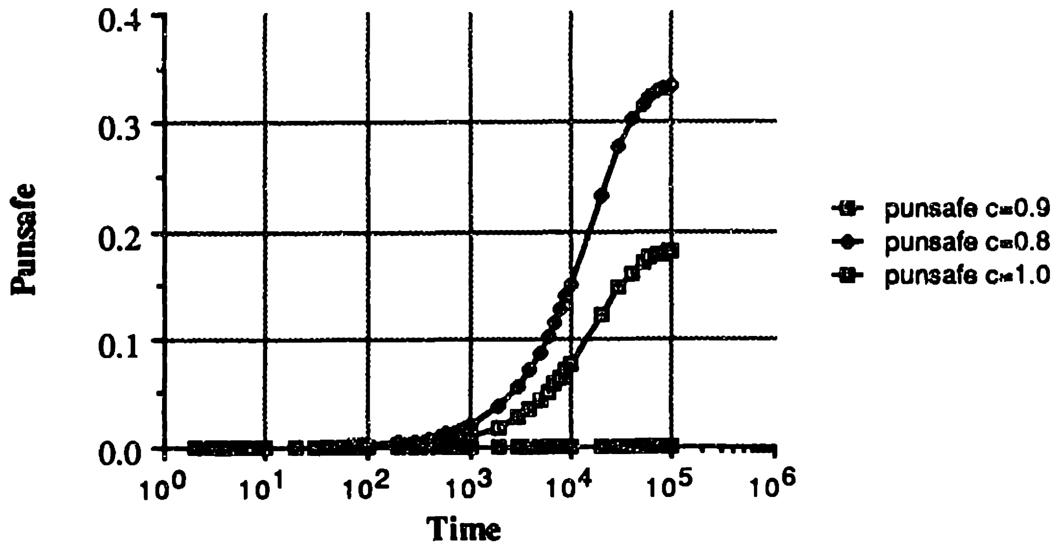


Figure 5.3: p_{unsafe} vs c For A Simplex Processor

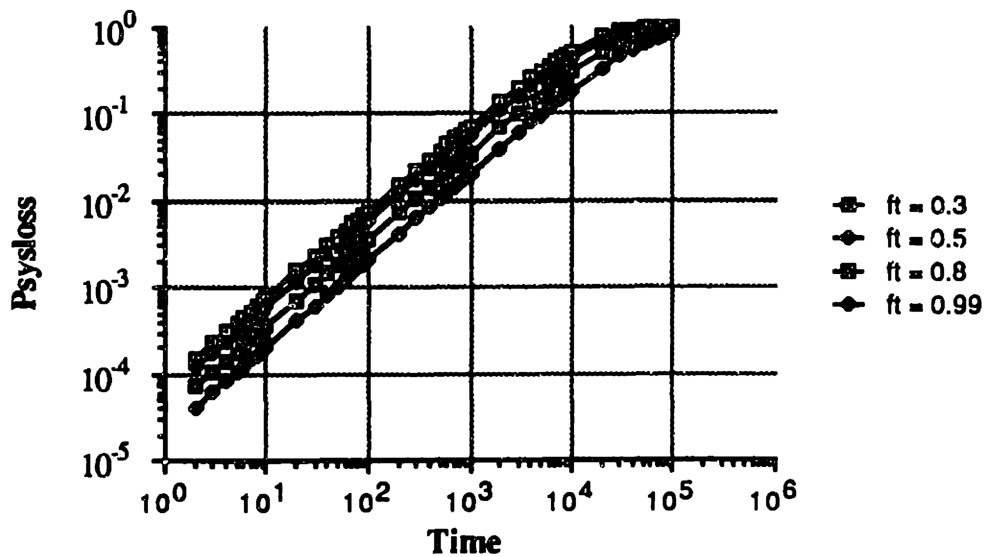


Figure 5.4: $p_{sysloss}$ vs f_t For A Simplex Processor

5.1.3 NEFTP Reliability Analysis

The Markov model for the NEFTP behavior depends on the redundacy scheme employed. We begin the analysis by assuming that the most conservative scheme is employed. Such a scheme, henceforth known as *redundancy scheme I*, would result in system shutdown after the first permanent NE failure. Since the NEFTP is a 1-Byzantine resilient architecture, any single failure can be covered with unity probability. Therefore, the parameter c is equal to 1.0 in the resulting model (Figure 5.5). The model consists of essentially two sub-groups of operative states, plus a shutdown and an unsafe failure state.

The first sub-group consists of states 1, 2, 3, and 5. State 1 is the *initial zero fault state*. A PE failure moves the system to state 2, the *PE failure reconfiguration state*, with unity coverage. Correspondingly there is a *NE failure reconfiguration state*, state 3. From state 1, any single NE failure will move the system to state 3 with unity coverage. If these failures are transient, the system will reconfigure back to state 1. If a second failure occurs in either reconfiguration state, a transition to the *unsafe system failure state* (state 10) occurs. The exception is if these "near-simultaneous" failures occur on behalf of the PE/NE pair that make up a single fault containment region. Such an occurrence moves the system to state 5, and does not result in unsafe failure. In state 5 there exists both a NE and a PE failure; it is assumed that the reconfiguration procedure will be such that the system will attempt to reconfigure the NE failure first. From either state 3 or 5, reconfiguration of a permanent NE failure results in transition to the *shutdown state*, state 11. This is because loss of a NE logically denotes loss of the corresponding FCR, and the resulting number of FCRs is insufficient to satisfy the cardinality requirement for minimal Byzantine resilience. From state 5, reconfiguration of a transient NE failure yields a transition back to state 2. If the processor failure that put the system into state 2 is permanent, the system will reconfigure to state 4.

State 4 begins the second sub-group (states 4, 6, 7 ,8, and 9) of the model. In state 4 there exist 3 working PEs and 4 working NEs. The system is still a 1-Byzantine resilient, fault masking architecture at this point; therefore a single, arbitrarily malicious failure can still be tolerated. The three processors provide the fault masking capability, and the 4 NEs provide the cardinality, connectivity, and synchronization for 1-Byzantine resilience. This is an *optional initial state* for a "reduced" NEFTP, hence the second sub-group of the model is similar to the first.

From state 4, a processor failure causes a transition to state 6. If this failure is transient, the system will reconfigure back to state 4. If it is permanent, the system will reconfigure to the *shutdown state*. While in state 6, another component failure would cause unsafe system failure to occur, again with the exception of a corresponding NE/PE pair failure. Such a failure would cause a transition to state 9. From state 4, a failure of any NE which still hosts a PE would result in a transition to state 8. If the failure is temporary, reconfiguration back to state 4 occurs, otherwise, reconfiguration to the *shutdown state* occurs. Occurrence of

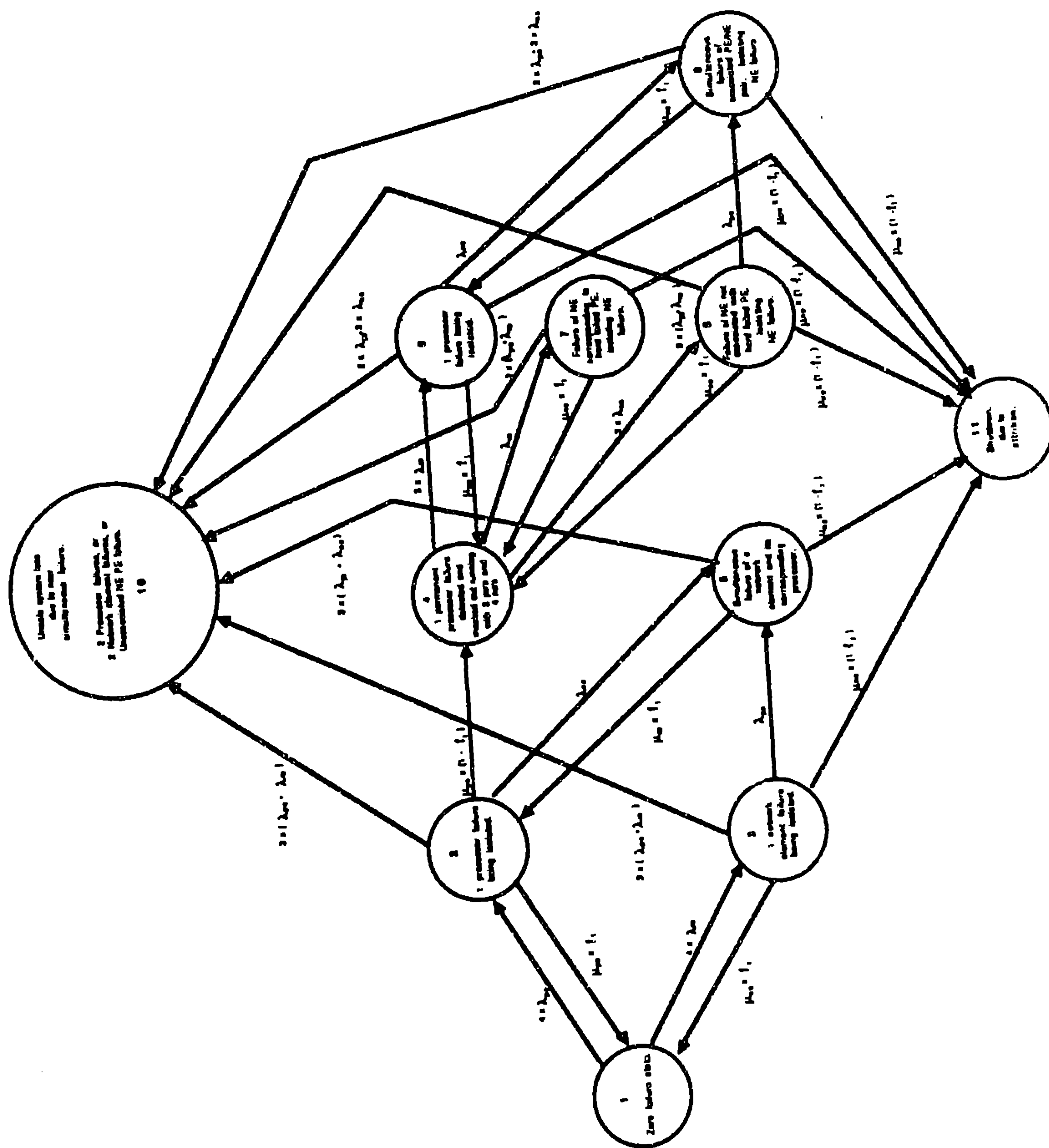


Figure 5.5: NEFTP Markov Model: Redundancy Scheme I

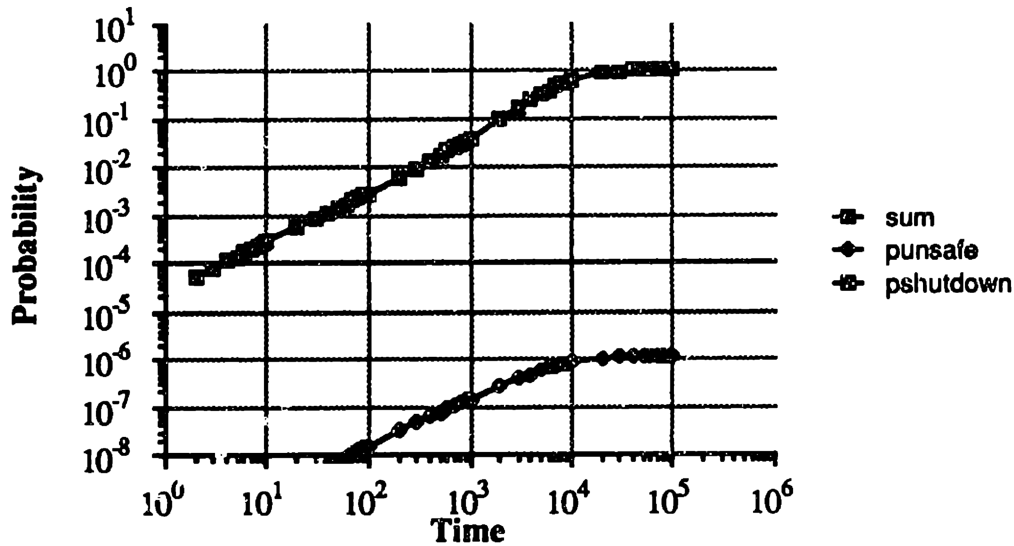


Figure 5.6: p_{unsafe}, $p_{shutdown}$, and $p_{unsafe}+p_{shutdown}$ For NEFTP Redundancy Scheme I

another failure while in state 8 would result in unsafe system failure except in the case of a corresponding NE/PE pair failure which would result in transition to state 9. From state 4, failure of the NE which does not host a PE results in transition to state 7. The analysis from state 7 is very similar to that of state 8 except that a corresponding NE/PE failure mode is not possible. The analysis from state 9 is very similar to that of state 5. In state 9, a NE and a PE failure exist. The system must try to recover the NE first. Successful recovery results in transition back to state 6, otherwise shutdown results. Any additional failures while in state 9 result in unsafe system failure.

In Figure 5.5 $p_{unsafe} = p_{10}$ and $p_{shutdown} = p_{11}$. A plot of these probabilities and their sum is shown in Figure 5.6.

Some interesting observations arise when viewing this plot. p_{unsafe} is 5 orders of magnitude less than $p_{shutdown}$. In Figure 5.6 the plot of $p_{shutdown}$ is coincident with the plot of the sum of $p_{shutdown} + p_{unsafe}$. This redundancy managment scheme is obviously inappropriate for applications of the ultra-critical category. For such an application the probability of system loss is not substantially less than that of the simplex processor; $p_{sysloss}$ at the end of a 10 hour mission is approximately $0.28 * 10^{-3}$. This is four orders of magnitude below the low end of the target reliability range. However, this redundancy managment scheme is suitable for critical applications where a backup to the computer is available. For such an application, $p_{sysloss}$ is $0.15 * 10^{-8}$ at the end of a 10 hour mission which is within the target range.

For ultra-critical applications a different redundancy scheme must be chosen

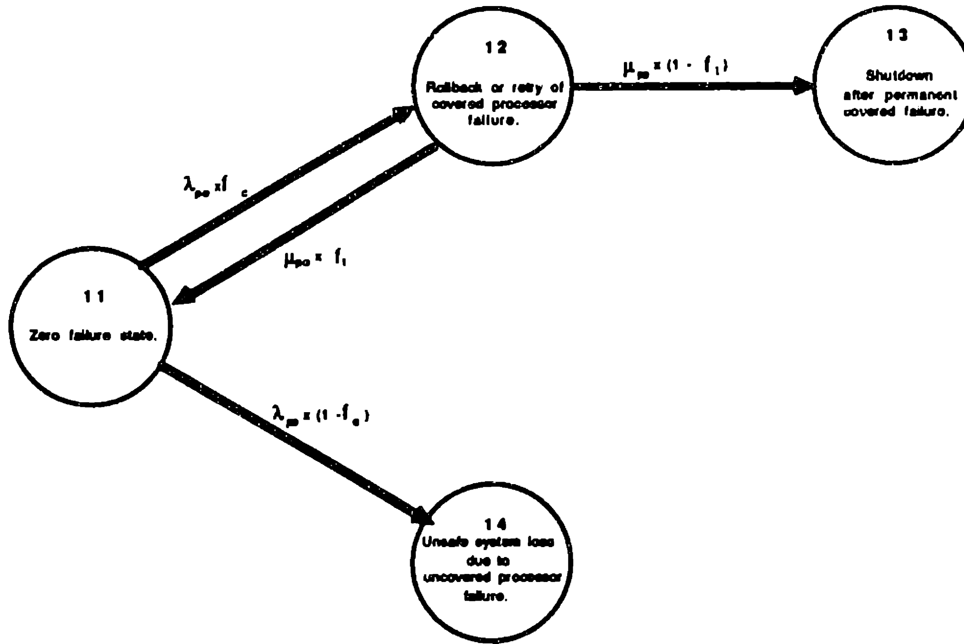


Figure 5.7: Addition To NEFTP Markov Model For Redundancy Scheme II

to reduce $p_{shutdown}$. For redundancy scheme I, the transition sequence state 1$\Rightarrow$ state 3$\Rightarrow$ state 11 (Figure 5.5) is the principal cause of this high probability of shutdown. It is NE "attrition" which results in the high value of $p_{shutdown}$. An ultra-critical application must utilize the substantial resources still functionning after one permanent NE failure.

As suggested above, a less conservative redundancy managment technique could be employed. Instead of shutdown occurring at state 11 (Figure 5.5), the system could automatically degrade to a simplex processor and continue operation. A model of this redundancy management scheme, henceforth known as *redundancy scheme II*, could be obtained by replacing state 11 of Figure 5.5 with the sub-model shown in Figure 5.7. Figure 5.7 is identical to the simplex processor model (Figure 5.1), except that the states have been renumbered. The interesting probabilities map onto this new aggregate model as follows: p_{unsafe} is equal to $p_{10} + p_{14}$, and $p_{shutdown}$ is equal to p_{13}. These probabilities along with their sum are plotted in Figure 5.8.

Figure 5.8 demonstrates several interesting points about this new redundancy management scheme. For ultra-critical applications $p_{sysloss}$ is equal to $0.85 * 10^{-7}$ at the end of a ten hour mission. This is within the range of target reliability requirements. However, for critical applications $p_{sysloss}$ is $0.29 * 10^{-7}$. This is an increase of almost an order of magnitude over that of redundancy scheme I. This is due to the fact that redundancy scheme II allows operation of the system in a non-Byzantine resilient mode. Operation in a non-Byzantine resilient mode results in a decrease of $p_{shutdown}$ and an increase in p_{unsafe}.

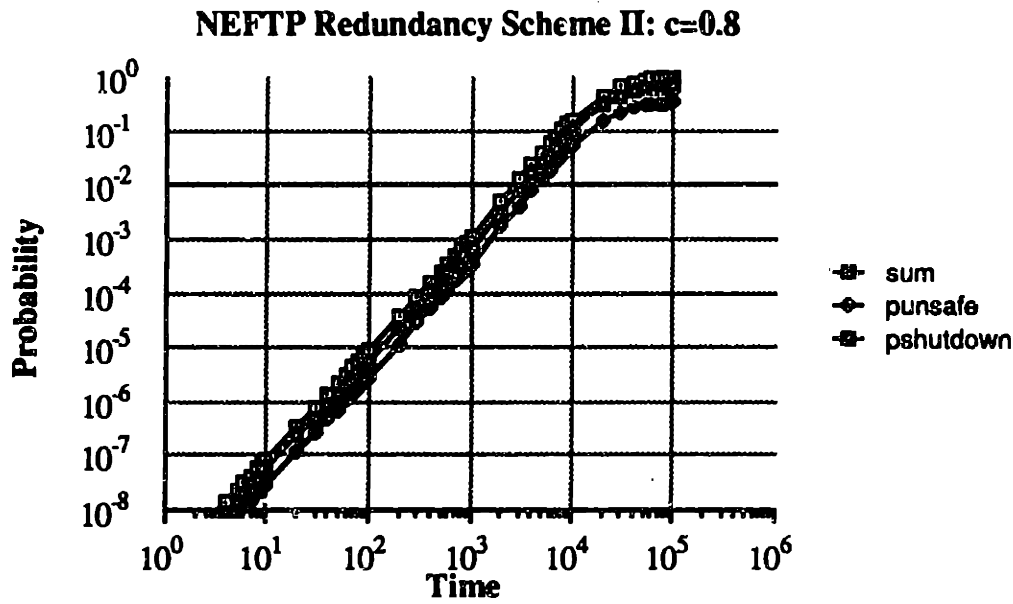


Figure 5.8: $p_{unsafe}, p_{shutdown},$ and $p_{unsafe} + p_{shutdown}$ For The NEFTP Redundancy Scheme II

Yet another redundancy managment scheme could be employed to achieve satisfactory results for both ultra-critical and critical applications. The desired strategy would attempt to use all the NEFTP resources left at the time the system enters the *shutdown state* of Figure 5.5. To do this the NEFTP would continue running at the maximum possible redundancy level until the occurrence of the next permanent NE failure, henceforth known as *redundancy scheme III.* A model for this scenario can be made by replacing state 11 of Figure 5.5 with the sub-model shown in Figure 5.9. State 11 (Figure 5.9) is one entry point of the sub-model, and is the state in which 3 FCRs are fully operative. It is possible to get to state 11 (Figure 5.9) from either state 3, 5, 7, or 8 of Figure 5.5. From state 11, covered NE failures move the system to state 13, a *NE failure reconfiguration state.* In state 11, the only type of uncovered NE failure which could occur is a Byzantine failure. Therefore the value for coverage that ought to be used for the transition from state 11 to state 13 is one minus the probability that the fault is Byzantine. This value would be very near unity. However, the same value for coverage will be used here that is used in the rest of the anaylsis, resulting in quite conservative results. If the failure is transient the system moves back to state 11. If the failure is permanent the system moves on to state 17, the *shutdown state.* Occurrence of another failure while in state 13 results in transition to the *unsafe system failure state*, state 18. From state 11, a PE failure moves the system to state 14, a *PE failure reconfiguration state.* All PE failures which occur in state 11 will be covered since it is impossible for a faulty PE to lie inconsistently through an operative NE. If the failure is transient

the system moves back to state 11. If the failure is permanent, the system moves on to the *shutdown state*. Occurrence of another failure while in state 14 causes the system to move to the *system fail state*. From state 11, uncovered NE failures result in transition to the *unsafe system failure state*. Another entry point is state 12 in which all NEs are operative, but only two PEs are operative. It is possible to get to state 12 from state 6 of Figure 5.5. From state 12, PE failures failures cannot be handled with unity coverage, though NE failures can. From state 12, NE failures move the system to state 15, a *NE failure reconfiguration state*. If the failure is transient the system moves back to state 12. If the failure is permanent the system moves on to state 17, the *shutdown state*. Occurrence of another failure while in state 15 results in transition to the *system fail state*, state 18. From state 12, a covered PE failure moves the system to state 16, a *PE failure reconfiguration state*. If the failure is transient the system moves back to state 12. If the failure is permanent, the system moves on to the *shutdown state*. Occurrence of another failure while in state 15 causes the system to move to the *unsafe system failure state*. From state 12, uncovered PE failures result in a transition to the *unsafe system failure state*.

It must be pointed out that certain aspects of the NE must be modified as the redundancy level is decreased. For example the current implementation of two round exchanges must be modified to tolerate a single non-Byzantine fault when only three NEs are operative. Currently the value received in the second round from the initial source FCR is not voted. Given a scenario where a *From A* exchange is performed by three NEs (A, B, and C), a single passive failure by NE B may result in each of the two NEs A and C voting two copies of the message - one of which is corrupted. The voter output will not be guaranteed. A modification to alleviate this problem would be to remove the second round source constraint when the NE redundancy level drops below 4.

For redundancy scheme III, p_{unsafe} is equal to $p_{10} + p_{18}$. $P_{shutdown}$ is equal to p_{17}. These probabilities along with their sum are plotted with respect to time (Figure 5.10). Upon viewing the plot, the desired results appear to have been achieved. Considering ultra-critical applications, the probability of system loss is low, $p_{sysloss} = 0.24 * 10^{-6}$ at the end of a 10 hour mission. This value of $p_{sysloss}$ is about three times as large as that resulting from redundancy scheme II. This can be explained by comparing the sub-model for redundancy scheme III (Figure 5.9) with that of redundancy scheme II (Figure 5.7). Due to the greater number of resources employed in redundancy scheme III, transitions to either the *shutdown* state or the *unsafe system failure* state (Figure 5.7) occur with three times the likelihood of tranisitons to either of those states in Figure 5.7. Because $p_{shutdown}$ dominates p_{unsafe}, increasing the value of coverage used does not help the ultra-critical case. Increasing the coverage will increase $p_{shutdown}$ and thus $p_{sysloss}$ for ultra-critical applications. Considering critical applications, $p_{sysloss}$ is equal to $0.14 * 10^{-7}$ at the end of a 10 hour mission.

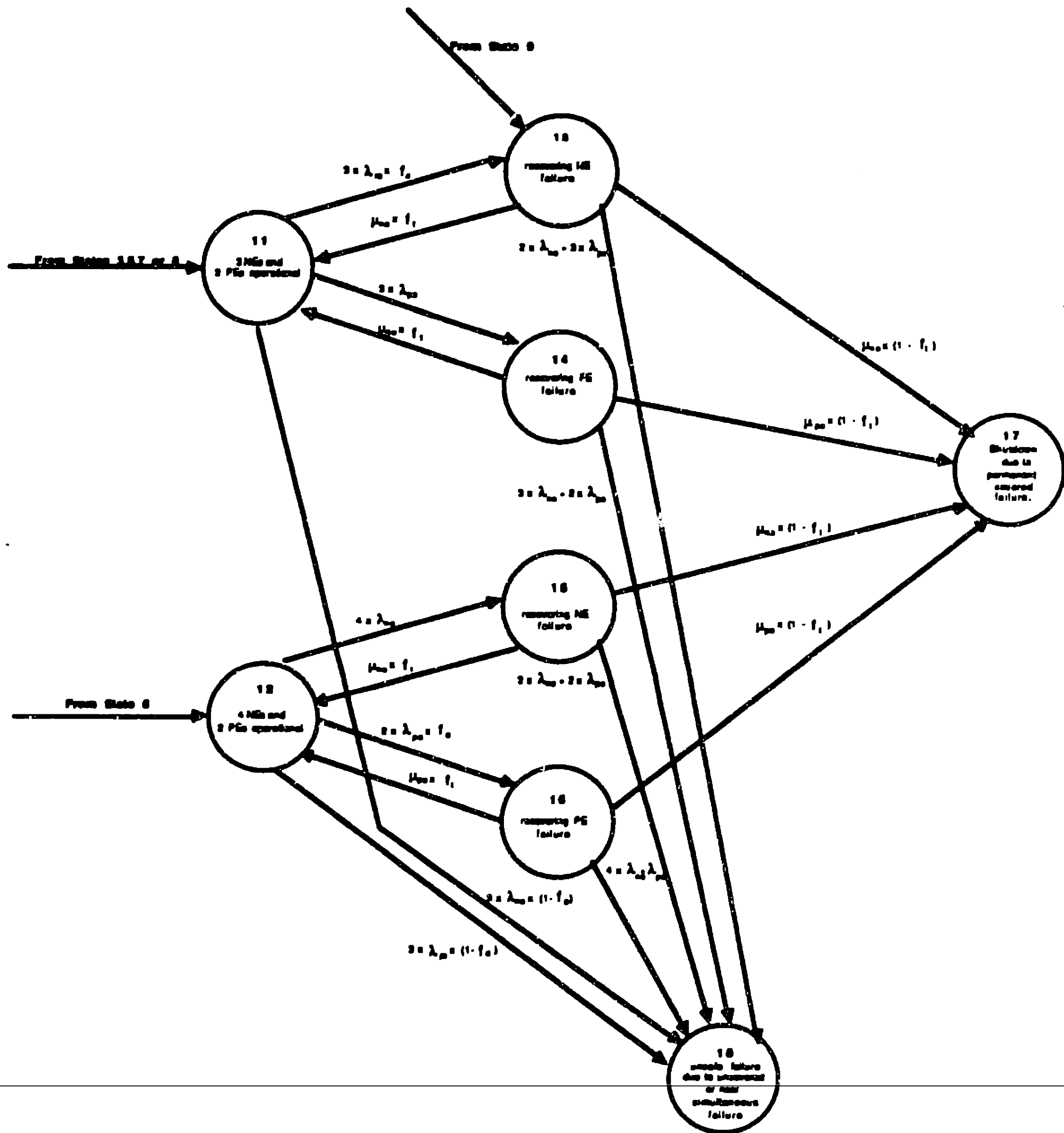


Figure 5.9: Addition to NEFTP Markov Model For Redundancy Scheme III

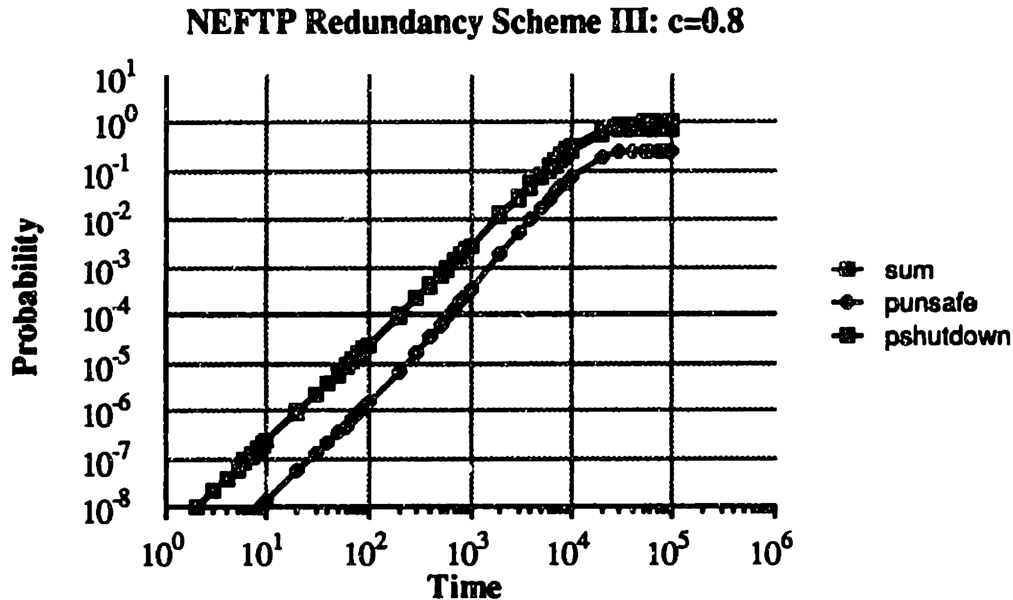


Figure 5.10: $p_{unsafe}, p_{shutdown}$, and $p_{unsafe} + p_{shutdown}$ For The NEFTP Redundancy Scheme III

What is the best that can be done? There are still resources available in the shutdown state of this last scenario. Will the reliability improve if one decides to continue running at this point instead of shutting down? To answer these questions, one must understand the relationship between p_{unsafe} and $p_{shutdown}$. Continued operation beyond the second permanent failure decreases $p_{shutdown}$ but increases p_{unsafe}. For critical applications where backups are available $p_{sysloss}$ equals p_{unsafe}. Hence maximum reliability is achieved when p_{unsafe} is minimized. Such applications would choose reliability managment scheme I. For ultra-critical applications where no backup exists $p_{sysloss}$ is equal to the sum of $p_{shutdown}$ and p_{unsafe}. Hence maximum reliability is achieved when their sum is minimized. However, the choice of redundancy managment schemes affects the two in opposite manners. In the three redundancy schemes considered, $p_{shutdown}$ dominates in all cases. Continued operation beyond the second permanent failure will reduce $p_{shutdown}$ and increase p_{unsafe}. The value of p_{unsafe} for scheme III is therefore a lower bound of the best p_{unsafe} that will be achieved by continued operation. Thus for ultra-critical applications, $p_{sysloss}$ is lower bounded by $0.14 * 10^{-7}$ at the end of a 10 hour missiton. It must be noted that this lower bound for ultra-critical applications is better than that which is achieved in scheme II. Scheme II had a lower value of $p_{shutdown}$ than did scheme III, but scheme II had a higher value of p_{unsafe}. Therefore, use of available redundancy may achieve a lower $p_{sysloss}$ for ultra-critical applications than is achieved by immediate reconfiguration to a simplex processor. A summary of the important probabilities for the various scenarios is shown in Figure 5.11.

	Simplex Processor	NEFTP Redundancy Scheme I	NEFTP Redundancy Scheme II	NEFTP Redundancy Scheme III
Probability of Unsafe Failure	2.00×10^{-4}	1.55×10^{-9}	2.96×10^{-8}	1.36×10^{-9}
Probability of Shutdown	4.00×10^{-4}	2.81×10^{-4}	5.62×10^{-8}	2.34×10^{-7}

Figure 5.11: Reliability Analysis Summary

5.2 Performance Evaluation

This section investigates the performance cost associated with the fault tolerance aspects of the NEFTP relative to that of other Byzantine resilient architectures such as SIFT, MAFT, and the AIPS FTP. The focus of this analysis will take the context of a typical fault tolerant computer application, namely flight control. In such an application both performance and reliability are critical. To understand the areas in which fault tolerance impacts performance, ignore for the moment reliability, and consider a flight control application implemented on both a fault tolerant processor and a simplex processor. Utilized as a flight control computer, a fault tolerant processor must perform several tasks. Among these are the tasks of self tests, scheduling, FDIR (Fault Detection, Isolation and Reconfiguration), and the control task itself. Given no reliability constraints, a simplex processor would perform only scheduling and the control task.

Self tests are motivated by a desire for reliability. Self tests are run for the purpose of uncovering latent faults in the system, and for the purposes of this discussion can be viewed as a cost of fault tolerance. The performance cost of self tests is dependent on the processor throughput and the complexity of the tests. The issue of processor throughput affects all aspects of performance. This illustrates the advantage of the NEFTP in utilization of off-the-shelf processors. As advances in industry increase the throughput of single board processors, the processors of the NEFTP can be readily upgraded to take advantage of the performance increase. However, this also illustrates the inherent advantage of parallel processing fault tolerant architectures such as MAFT. The design of a NE based architecture incorporating parallel processing is under development [Har87]. Another factor of interest here is the complexity of the tests, which depend on the complexity of the hardware being tested. All fault tolerant architectures must be designed for minimum hardware complexity and maximum testability. Realistic measures of the complexity of the hardware of varying fault tolerant architectures are difficult to achieve due to the lack of information in the literature. The NEFTP has been successful in this regard. The Debug Command Language of the NEFTP is intended to form the basis of self tests with very high coverage and resolution[2]. The NEFTP is designed for efficient

[2]Work is currently being performed on the self test capability of the NE with moderate success.

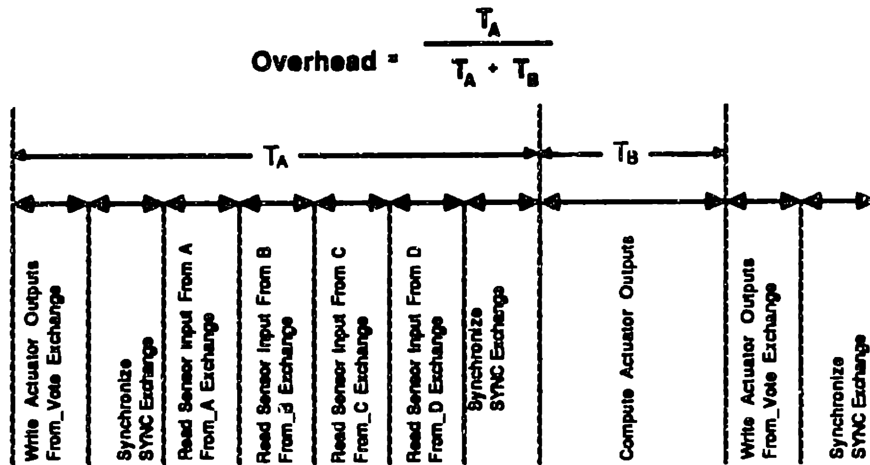


Figure 5.12: Typical Control Loop

testability.

Scheduling must be done whether the architecture is fault tolerant or not. The author is not convinced that fault tolerance necessarily impacts the complexity of scheduling in a seriously negative fashion. As a real-time system, a flight control computer must implement context switching in an efficient manner. A simplex machine could implement a clock based, interrupt driven rate group scheduler, or a task driven scheduler. The AIPS FTP implements a rate group scheduler. The essential increase in complexity required is that the interrupts be exchanged among FCRs. The NEFTP implements a task driven scheduler with no inherent increase in complexity over a similar scheduler for a simplex machine. The MAFT scheduler is significantly more complex, but this is due, in part, to the fact that it is a true multiprocessor and not simply because it is Byzantine resilient. Therefore in this analysis, scheduling is not viewed as a cost of fault tolerance.

FDIR is clearly a cost of fault tolerant architectures which allow reconfiguration around permanent failures. FDIR is typically either *passive* or *active*. Passive FDIR involves analyzing the syndrome information generated from the data exchanges required of the application. However this information need not be identical on all FCRs and hence must be distributed via an interactive consistency exchange before analysis may begin. Active FDIR involves executing additional data exchanges designed to generate syndrome information that will help isolate the fault. The important factor here (aside from processor throughput) is the throughput of the Byzantine resilient data exchange mechanism.

Fault tolerance impacts the performance of the application control task as well. A typical flight control application involves a loop of reading sensor input, performing computation on this input to determine actuator commands, and then writing the output to the actuator. A fault tolerant flight control system typically uses redundant sensors and actuators. Consider the case where there are four FCRs,

four sensors, and four actuators. The control loop implemented on such an architecture would consist of four interactive consistency exchanges of the sensor input, followed by computation, followed by an output consensus exchange of the actuator commands. Such a loop, described in NEFTP terminology, is shown above (Figure 5.12). The overhead induced by the fault tolerance is again the time required for the data exchange.

The throughput of the Byzantine resilient data exchange mechanism is an important factor affecting the performance cost of fault tolerance. Given architectures of comparable testability and processor throughput, relative efficiency is determined by the throughput of the fault tolerant data exchange mechanism. The following is an effort to compare the data exchange throughput of some Byzantine resilient architectures.

5.2.1 NEFTP Analysis

The following are measurements of the NEFTP system and sub-systems currently under test[3]. The NE of the system is operating with a nominal data link speed of 64 Mbps. The PE of the system is a Motorola 68020/68881 combination with a 12.5 MHz system clock. All PE code is written in C using the Green Hills' C compiler.

The Data Exchange Parameters

The time the NE requires to perform the various message exchanges is fundamental to the overall NE data exchange throughput. This time will be refered to as *NE time* (Figure 5.13). The NE time for a given exchange is considered to be the time between the processor's write to the class FIFO and the last of the NE's writes to the REC FIFO. This time, as viewed from any PE in the system, is dependent on three factors.

1. The exchange class.
2. The length of the message being exchanged.
3. The processor skew.

The processor skew is a function of synchronization frequency and will be discussed later. The NE times have been measured from the slowest processor's class FIFO write and reflect the true NE times for the case of zero processor skew. Once processor skew is known (see below) it can be added to these NE times to arrive at the NE times as viewed from the fastest PE. Even in the case of zero PE skew, the NE requires between one and two SERP exchange times before a message exchange will be honored. This is what accounts for the high, low and medium times listed

[3]The measurements below are the results of tests currently being conducted by Ross Dettmer.

Exchange Type	Size (Bytes)	Exchange Times (μs) [Throughput] (Mbps) low	med	high
SYNC	NA	6.5	9.75	13.0
1 Round	16	11.5 [11.1]	14.75	18.0 [7.11]
	240	67.5 [28.7]	70.75	74.0 [25.9]
2 Round	16	14.0 [9.14]	17.25	20.5 [6.24]
	240	98.0 [19.6]	101.25	104.5 [18.3]

Figure 5.13: NE Data Exchange Times and Throughput

(Figure 5.13). In the high case, the class FIFO access occurs immediately preceeding the start of a SERP exchange; in the low case, it occurs immediately after the start of a SERP exchange.

Figure 5.13 also lists a throughput measure for each entry in the exchange time table. This throughput corresponds to the amount of PE data exchanged per unit time if the given exchange was performed repeatedly.

Processor Skew

The issue of processor skew is critical to many aspects of the NE. PE skew is reduced by periodic synchronization. PEs synchronize by sending a message and then busy waiting until it is returned by the NE. The maximum processor skew is a function of the frequency of synchronization and the post-synchronization skew. The PE post-synchronization skew is a function of the NE post-synchronization skew plus the busy wait loop length. The post-synchronization skew of the NEs is no

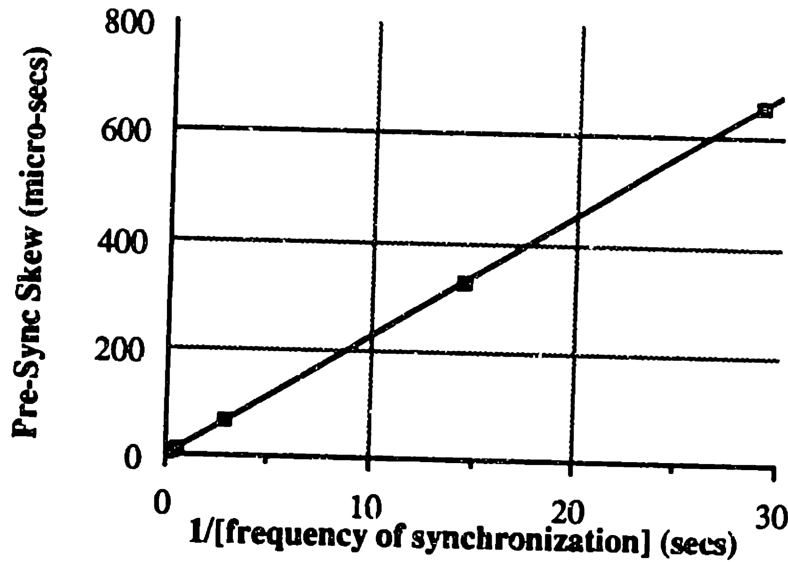


Figure 5.14: Processor Pre-Synchronization Skew as a Function of Synchronization Frequency

greater than $375ns$ as dictated by the FTC design. The PE pre-synchronization and post-synchronization skew are measured for various frequencies of synchronization.

The PE post-synchronization skew is measured as the difference in time between the REC FIFO reads of the synchronization message of different PEs. The PE pre-synchronization skew is measured as the difference in time between the class FIFO writes of the synchronization class byte of different PEs. The frequency of synchronization is varied by delaying a programmable time between synchronization exchanges. The post-synchronization skew is independent of frequency, with a maximum value of $2\mu s$. Pre-synchronization skew is plotted as a function of frequency (Figure 5.14).

The pre-synchronization skew should equal the post-synchronization skew plus the processor drift. For the large values of pre-synchronization skew, the post-synchronization skew is insignificant. Thus the pre-synchronization skew is effectively the processor drift. Processor drift develops because of their oscillator drift and non-clock determinacies in the PE hardware. The program utilized in this experiment does not exercise any area of non-clock determinacy other than the case of ram accesses involving parity errors. Therefore the skew should be dominated by oscillator drift. The value of drift computed from Figure 5.14 is approximately $2 * 10^{-5}$ which is plausible for commercial oscillators and the accuracy of the tests. Figure 5.14 provides the information necessary to determine what value of scoreboard timeout is applicable given the frequency of synchronization. The maximum scoreboard timeout in the current NE is 120 ms. Clearly very low frequency of

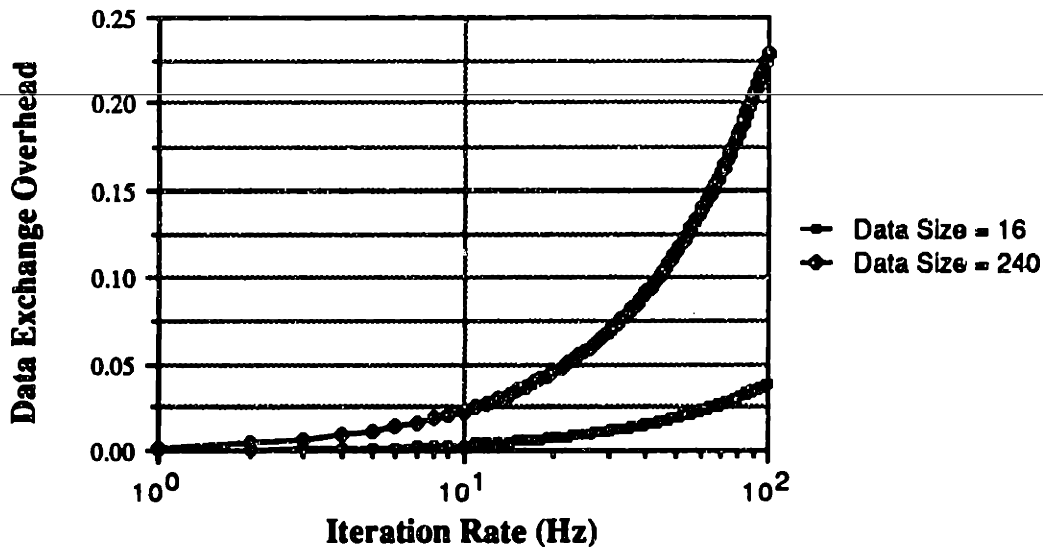


Figure 5.15: Fault Tolerance Related Data Exchange Overhead as a Percentage of Control Loop Length

synchronization can be tolerated in the current architecture.

Control Loop

A control loop like that shown in Figure 5.12 was programmed using simplified versions of the data exchange primitives. The primitives were simplified in that the ability to handle variable packet sizes and multiple packet messages was removed. The message queuing structures and error checking were also removed. The "computation" performed was merely a busy wait. A measure of data exchange overhead was computed as follows: $overhead = (data\ exchange\ time)/(total\ loop\ time)$. Values of this overhead were computed for several different "computation" times and two different packet sizes. The results are plotted in Figure 5.15.

In the current implementation, the overhead is comprised almost entirely of processor time. A control loop that uses a packet size of 16 bytes spends $386\mu s$ performing the fault tolerance related exchanges. An identical loop that uses the packet size of 240 bytes requires $2.2ms$ to perform the data exchanges. The data exchange time for the latter case can be broken down as shown in Figure 5.16. The time spent waiting for NE is $70\mu s$ (on average), which is roughly 3% of the time required for the data exchanges.

The conclusion to be drawn is that the NE is too fast for the current PE. The current implementations of the data exchange primitives require $3.5\mu s$ to perform a longword write to the NE. Enhancing the functionality of the data exchange

EVENT	TIME REQUIRED (μs)
PE writes From_A Exchange to the NE.	213
PE writes From_B Exchange to the NE.	213
PE writes From_C Exchange to the NE.	213
PE writes From_D Exchange to the NE.	213
PE reads the results of the exchanges above.	852
PE writes SYNC Exchange to the NE.	6
PE waits for the NE to process SYNC Exchange.	10
PE reads the results of the SYNC Exchange.	5
PE performs control law calculations.	Variable
PE writes From_Vote Exchange to the NE.	201
PE waits for the NE to process From_Vote Exchange.	50
PE reads the result of the From_Vote Exchange.	213
PE writes SYNC Exchange.	6
PE waits for the NE to process SYNC Exchange.	10
PE reads the results of the SYNC Exchange.	5
PE loops back to beginning.	12
Total Data Exchange.	2,222
PE waits for NE	70

Figure 5.16: Breakdown of Fault Tolerance Related Data Exchange Overhead When Using Maximum Packet Size

primitives will further increase PE/NE transaction times. Clearly the data exchange primitives need to be fully optimized to take advantage of the NE throughput. For example, the time required to write the four interacive consistency exchanges could be quartered if the PEs were not required to write null packets for the exchanges they did not source. This would necessitate a minor change in the NE global controller code which implements the two round exchanges. However, it may also require that packet sizes used in these exchanges be identical. The efficiency of the data exchange primitives is currently the limiting factor in the data exchange throughput of the NEFTP ensemble.

Comparison of Data Exchange Throughputs of Various Byzantine Resilient Architectures

The performance cost of implementing fault tolerance in the SIFT architecture is large [PB86]. The nominal speed of the inter-FCR communication link for SIFT is 4 Mbps. The corresponding number for the NEFTP is 64 Mbps. A single word broadcast can take anywhere from 8.6 to 18.2μs. A 5-way vote takes 413μs in the absence of errors. The resultant single round exchange throughput is 38.7 Kbps (thousand bits per second). The NEFTP single round exchange throughput ranges from 7.11 Mbps to 28.7 Mbps (Figure 5.13). SIFT requires 12.9ms to perform an interactive consistency exchange of 21 words - a throughput of 3.25 Kbps. The two round exchange throughput for the NEFTP ranges from 6.24 Mbps to 19.6 Mbps. This illustrates the severe penalty of implementing fault tolerance related data exchange via software tasks. The data exchange throughput of the NEFTP is considerably faster.

Little information is available concerning the throughput of the MAFT data exchange. The "effective" bandwidth of the MAFT data exchange network is 1 Mbps [KWFT88]. This would correspond to the data rate for single round exchanges, with two round exchanges occurring at half this rate, generously assuming the effective bandwidth includes the time required to vote values. The figures for the NEFTP compare favorably. The NEFTP data exchange throughputs are approximately an order of magnitude faster.

The latest AIPS architecture FTP has a data exchange throughput of 14 Mbps for two round exchanges. All exchanges are two round exchanges. Considering a control loop application like that of Figure 5.12, two round exchanges occur 4 times as often as one round exchanges. Weighting the NEFTP data exchange throughputs in this manner the resulting throughput would range from 6.41 Mbps to 19.8 Mbps. The NEFTP can be faster than the AIPS FTP if the maximum packet size can be used. In worst case, the NEFTP is 0.45 times as fast as the AIPS FTP.

5.3 NEFTP Evaluation Summary

Analytical modelling of the NEFTP demonstrates that it can provide the reliability required for critical and ultra-critical applications (approximately 10^{-9} and 10^{-8}, respectively). Performance evaluations of the NEFTP demonstrate that the data exchange overhead of fault tolerance is low compared to that of other Byzantine resilient architectures.

Chapter 6
Conclusions and Recommendations

The primary conclusion of the above analysis is that the NEFTP architecture does indeed meet the reliability requirements in an efficient manner. The NEFTP is reliable and efficient[1]. The compact design of the NEFTP, minimal interconnect, low power dissipation[2], and flexibility of processor implementation make the NEFTP an appealing architecture.

There is much future investigation relating to the NEFTP that should be considered. Substantial improvements over the current implementation can no doubt be realized. These areas of future work are discussed below.

The NEFTP is capable of performing simplified control applications at very high iteration rates. However, in order to demonstrate the legitimacy of the NEFTP a more realistic workload must be constructed including self tests, the FDIR mechanism, and a scheduler. Development of these must be an area of future work if the architecture is to be taken seriously.

In the current implementation of the NEFTP the processor is too slow to utilize the full bandwidth of the NE data exchange mechanism. As processor efficiency is increased, the NE data exchange throughput may need to be increased. There is opportunity for such improvements in the current implementation of the NE. The nominal bandwidth of the inter-FCR communication links is 64 Mbps, but the NE data exchange utilizes only between 9% and 45% of this. Without the SERP overhead, and the overhead of voting the data exchange bandwidth would be 50% to 100% of the nominal communication link bandwidth. It is plausible that a 30% reduction in the SERP overhead could be achieved by optimizing the corresponding global controller code. The current version of the SERP exchange code uses a frame size of 48 bytes, most of which is idle time. However, a previous version was coded that used to 16 byte frames back to back. This earlier version did not work correctly in the case were the minimum FTC wait occurred between frames; nonetheless, it illustrates that a minor improvement in SERP code effeciency may yield big gains. Additionally, for single round exchanges, the time required to vote and deliver messages could be substantially overlapped with the exchange of the message through a slight increase in complexity of the global controller. Currently voting of a message does not begin until after the entire message has been received. The minimal requirement is that voting not begin until after the first byte of the message has been received from all FCRs. Since voting and message exchanges occur at the same rate, the integrity of data would be maintained. The result would

[1]The efficiency of the NEFTP compares favorably to other fault tolerant uni-processors.

[2]The prototype NEFTP consumes approximately 200 Watts.

be that single round exchanges would take on the order of one FTC frame (not including SERP overhead) for exchange, voting and delivery instead of the current two. This would be an appreciable increase in throughput. The cost would be the complication of the global controller required to perform message exchange and voting and delivery simultaneously for single round exchanges. This optimization is not possible for two round exchanges due to the physical constraints imposed by the data path FIFOs.

Current efforts investigating the feasibility of reducing the NE functionality into a chip set or wafer scale project should be continued. The indicated disparity between the NE throughput and the efficiency of the PE data exchange primitives prove to be interesting in this context. In particular, this information impacts the utility of the multi-packet capability of the current NE. The multi-packet functionality was provided in order that PE/NE exchange overhead could be overlapped to some extent with the NE data exchange overhead. A multi-packet NE would allow the PE to write many packets before reading any, like WriteWriteWriteReadReadRead. A single packet NE would require the processor to read each packet after it was written, like WriteReadWriteReadWriteRead. Measurement of the speedup provided by the multi-packet NE finds it to be minimal in the current system. This is due to the fact that the time required by the PE to transition from completing a write to initiating a read in the single packet scenario is comparable to the time required by the NE to perform the exchange. This implies that even in the single packet scenario the PE is not delayed significantly by the NE data exchange time. Interestingly enough the PE/NE exchange overhead must be substantially reduced before a multi-packet NE produces a substantial speedup over a single packet NE. The important realization is that the multi-packet functionality of the NE can be removed without adversly affecting the performance of the current NEFTP. In fact some NE throughput could be sacrificed by reducing the width of the NE data paths, resulting in a more balanced PE/NE pair with no great performance loss. All this is good news when considering how to reduce the NE physical size. Efforts to reduce the NE should concentrate on a single packet NE with narrow data paths.

The NEFTP provides an interesting opportunity to perform experiments in design diversity. Hardware and software design diversity should be explored. To the extent that such diversity satisfies the constraint of bit for bit congruence, the NEFTP provides a very amenable test-bed for experimentation. The NEFTP can also be utilized for design diversity experiments which do not satisfy the bit for bit congruence constraint. The NEFTP is albeit less well suited for such experiments than architectures that employ approximate agreement voting algorithms in hardware like MAFT. However, the NEFTP is certainly capable of implementing design diversity that necessitates approximate agreement. Such an experiment would not use the *From Vote* exchange. All data must be exchanged via interactive consistency exchanges. The approximate agreement "voting" algorithm could be implemented

in software in the PEs. The overhead increase would be large. A "vote" exchange would require four interactive consistency exchanges plus considerable PE computation time instead of merely a single round exchange. Conducting such experiments will help determine if reliability is increased sufficiently (if at all) to balance the performance cost and/or increased hardware cost of approximate agreement.

Additionally, the arena of multi-processing can be explored to a limited extent using the NEFTP. In particular a two processor per FCR architecture, as currently utilized in the AIPS design, may be interesting. The "processor" of an AIPS FTP FCR is really two processors. One is the control processor (CP) and the other is the I/O processor (IOP), both of which communicate via a shared bus that hosts shared resources including the data exchange mechanism. Ordered communication between the CP and IOP is maintained via interrupts. In the current implementation, the CP runs the application task and the IOP performs sensor and actuator I/O as well as fault tolerance related data exchange. Such an architectual modification may prove beneficial to the NEFTP when real sensor/actuator I/O becomes an issue. Such a modification is planned as a logical expansion of the NEFTP. Each FCR has room for an additional processor. Polling must be used instead of interrupts to coordinate the communication between the two processors.

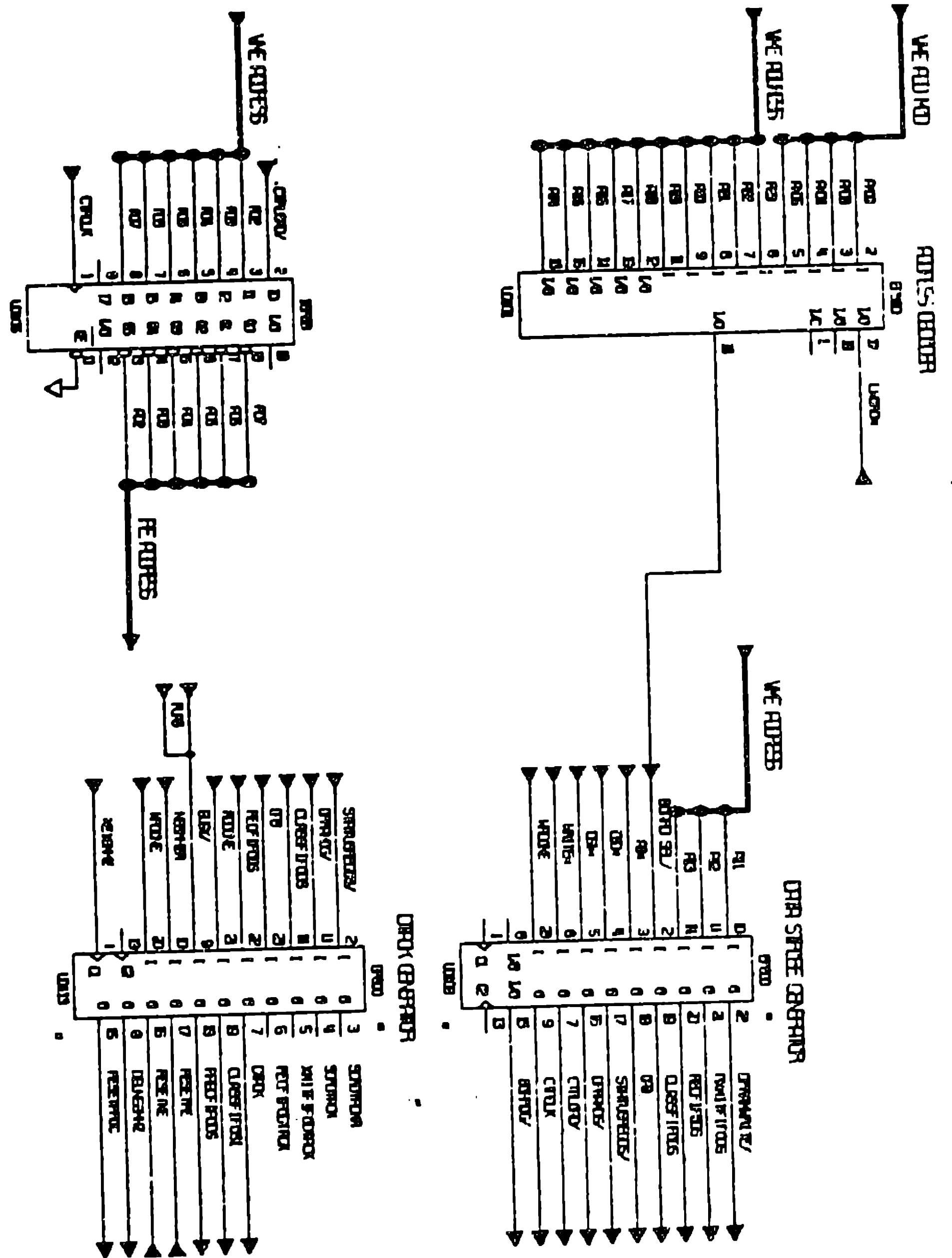
A NE is currently being developed for a full parallel processing architecture - the Fault Tolerant Parallel Processor (FTPP). This NE is significantly larger and more complex than that of the NEFTP. By utilizing the NE of the NEFTP as a single packet NE, it may be possible to support parallel processing with minimal modification. The modification would invlove redefining the PE/NE flow control information generated by the NE. Altering the meaning of Clear To Send to "NE Available" may be sufficient. A NE would be available if both its XMIT and REC FIFOs were empty. This must be accompanied by the constraint that a given PEs access of the NE must be separated by more than the maximum allowable processor skew in order to maintain the ordering of messages into the NE. The effective throughput of the NE as seen from any processing ensemble will decrease, perhaps tremendously. However, this may be offset by the increase in processing throughput.

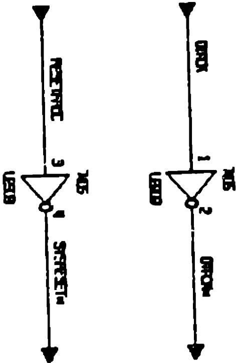
Finally, the reliability of the NEFTP could be boosted by utilizing the available space in each FCR for standby redundancy. Modelling can be conducted to determine if the spares should take the form of NEs or PEs. Another question is should the spares be running and waiting to take over operation (i.e. hot spares), or powered off (i.e. cold spares)? Again, analysis should be conducted to determine how each scenario affects reliability. Implementing stand by redundancy creates engineering difficulties. For example, how do cold spares get powered on? Can a stand by element fail in such a way as to corrupt the primary element? Furthermore, how is the failed element disabled so that it does not interfere with a recently activated spare? The NEFTP is amenable to such experiments since it has the space available in each FCR and at least spare PE should be easy to acquire.

In the author's opinion, the NEFTP succeeds as a useful vehicle to further explore many aspects of fault tolerant computing.

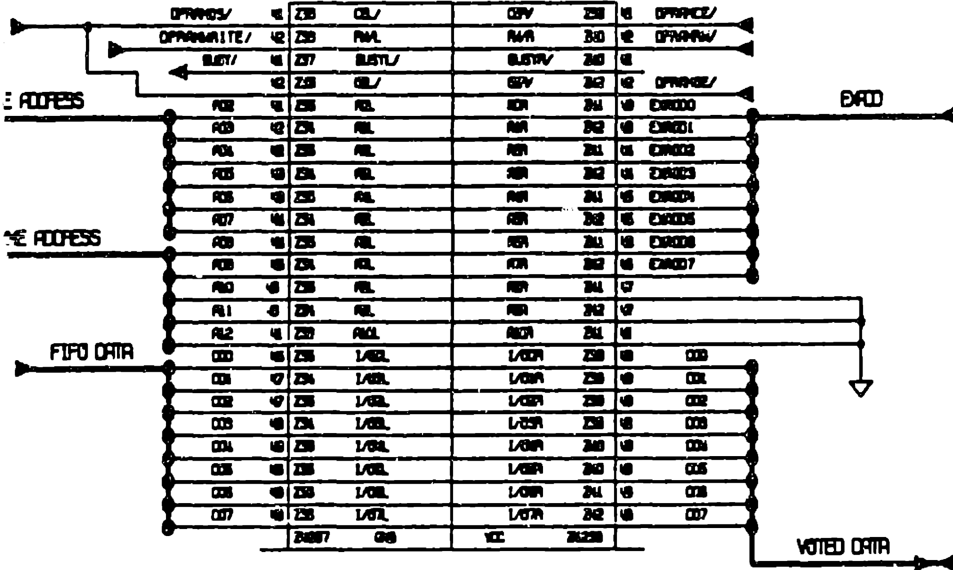
Appendix A
NE Schematics

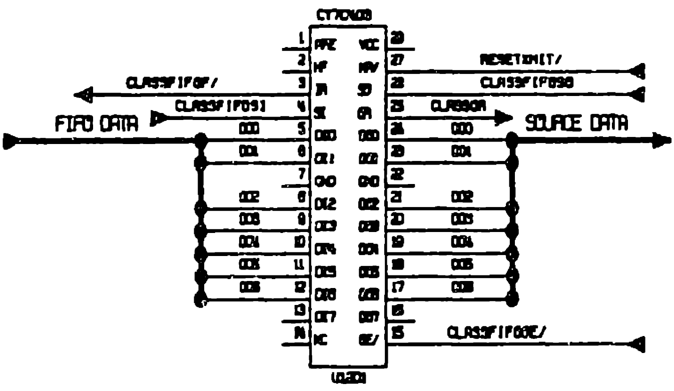
A.1 Sheet 1

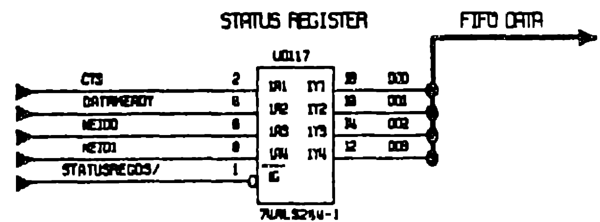




A.2 Sheet 2

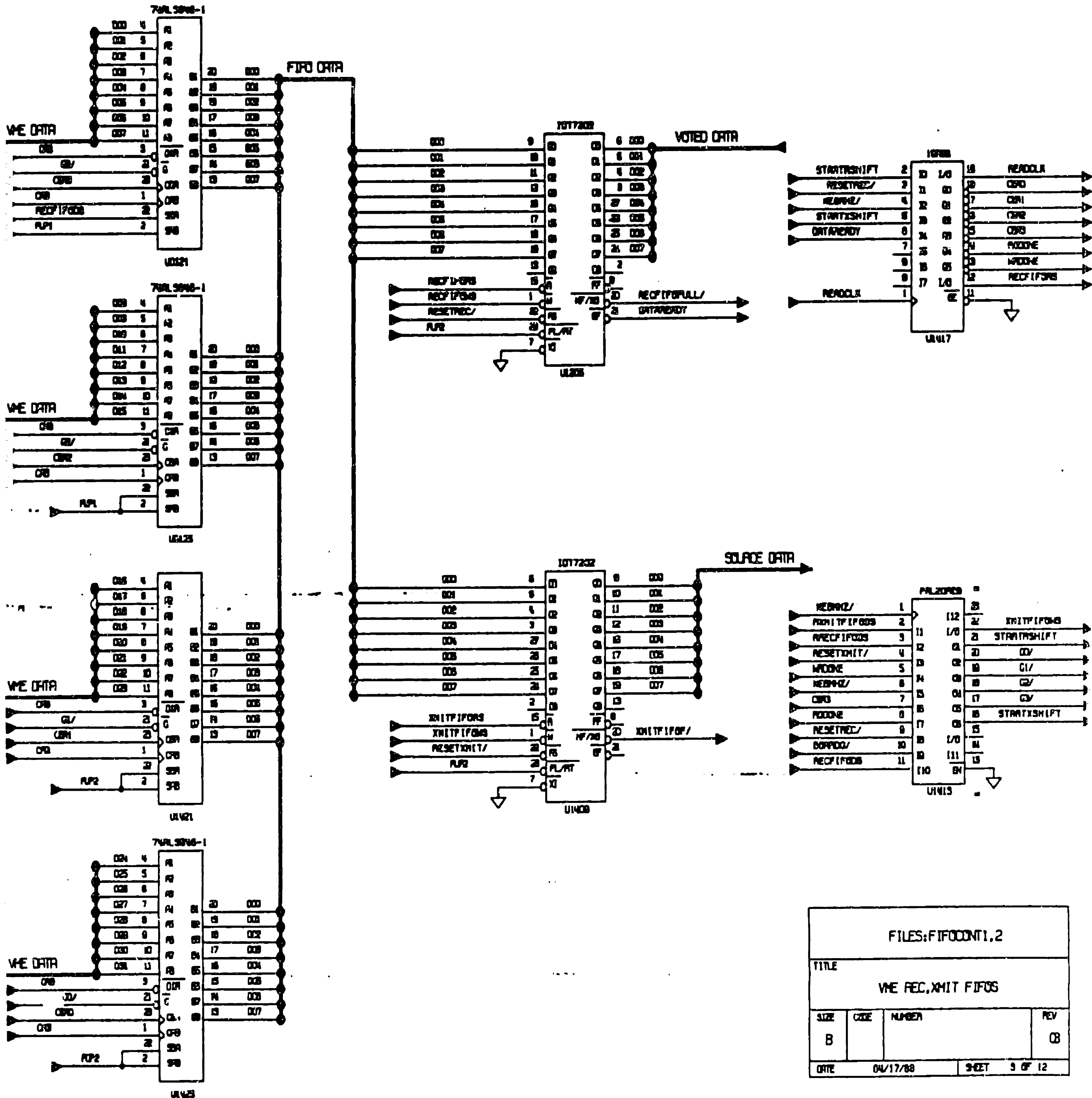




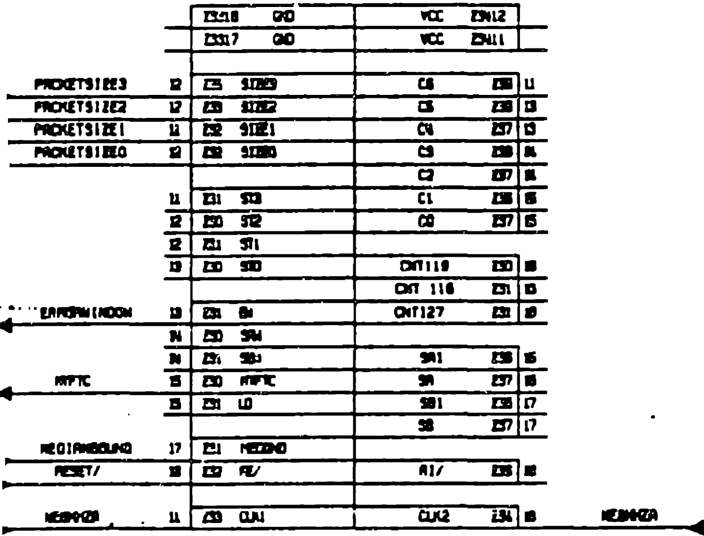


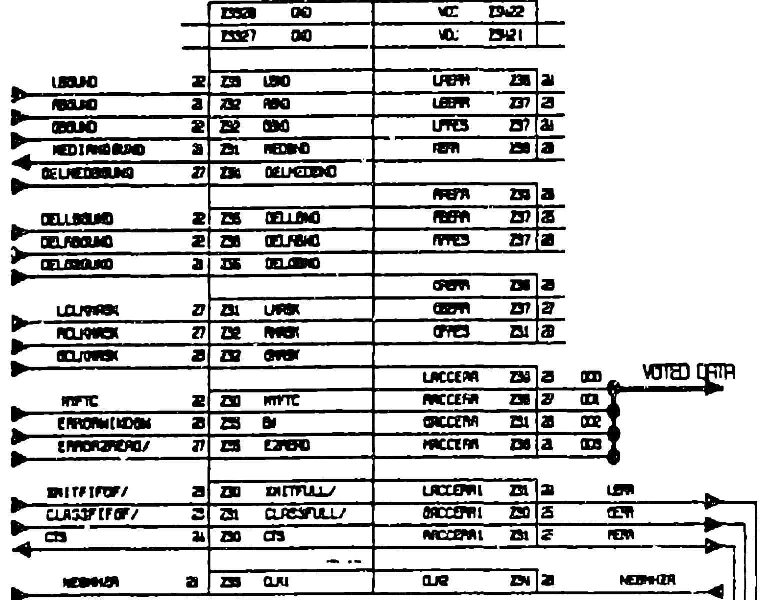
FILES:			
TITLE VME DPRAM,CLASS FIFO,STATUS REG			
SIZE B	CODE	NUMBER	REV 03
DATE 04/17/88		SHEET 2 OF 12	

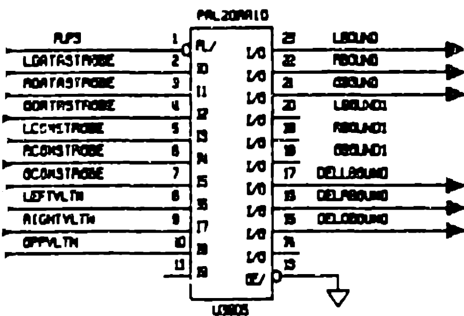
A.3 Sheet 3

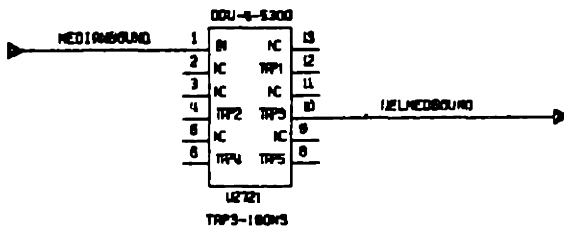


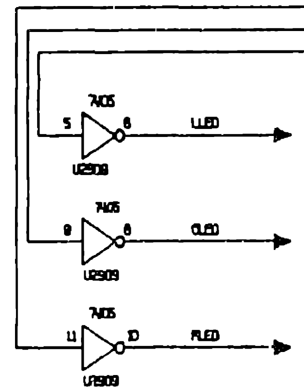
A.4 Sheet 4

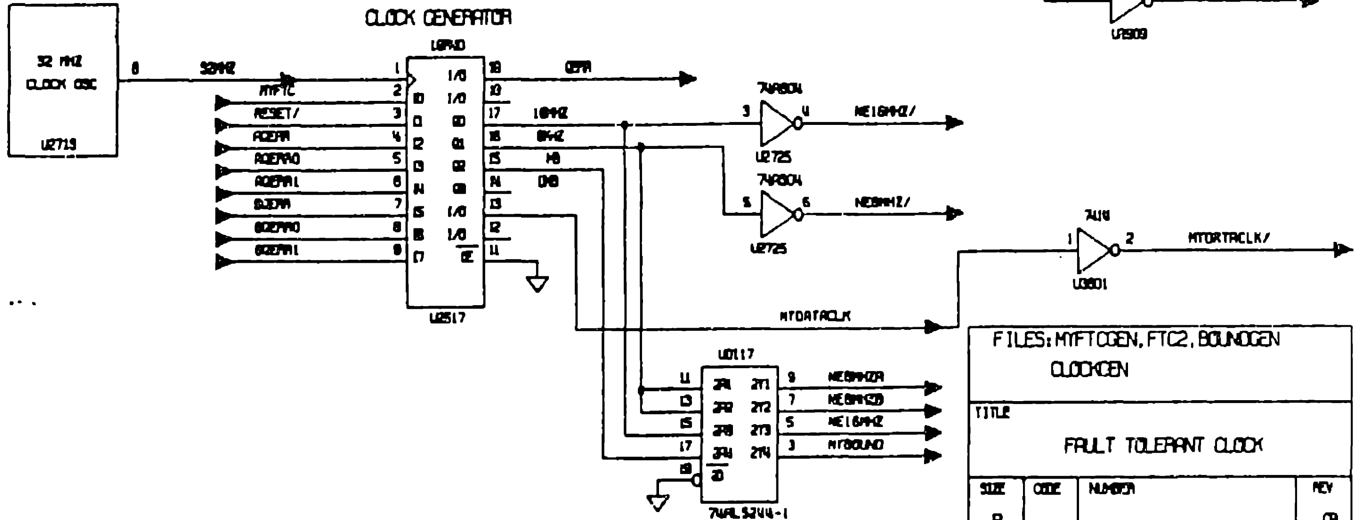




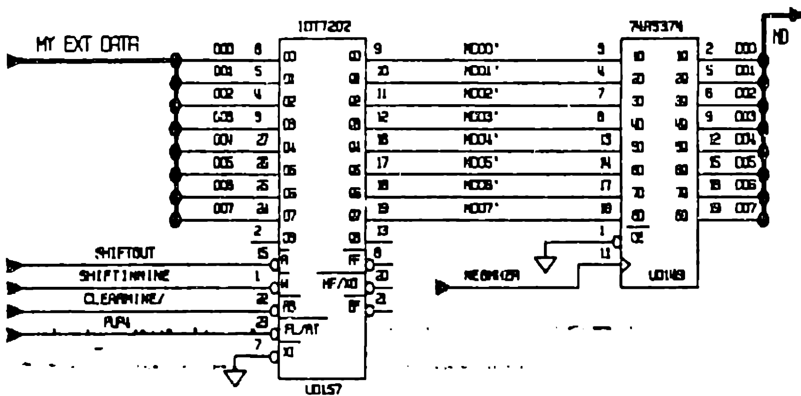


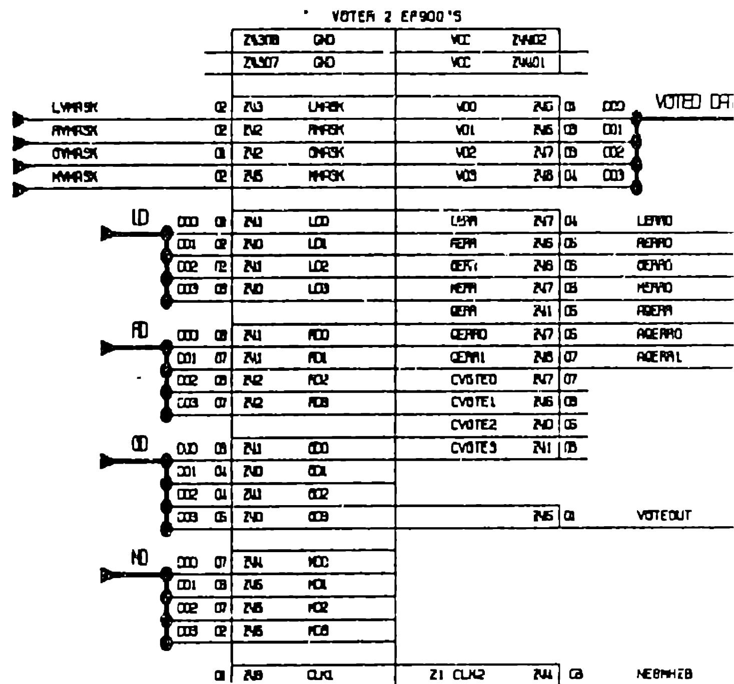


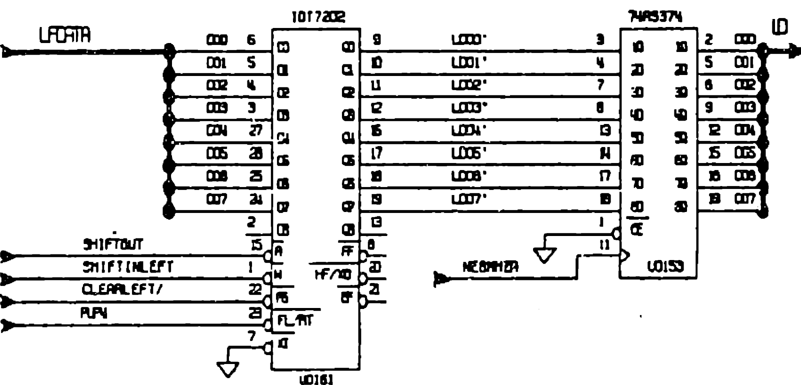


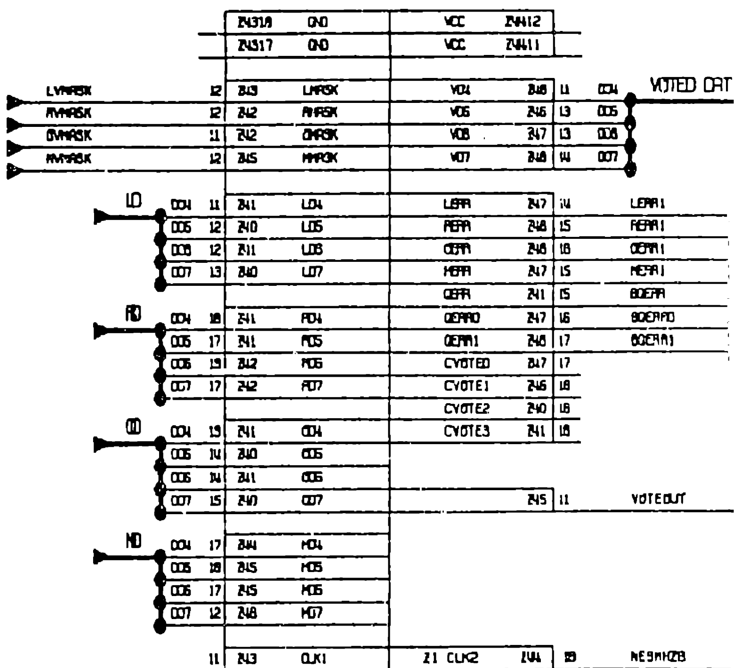


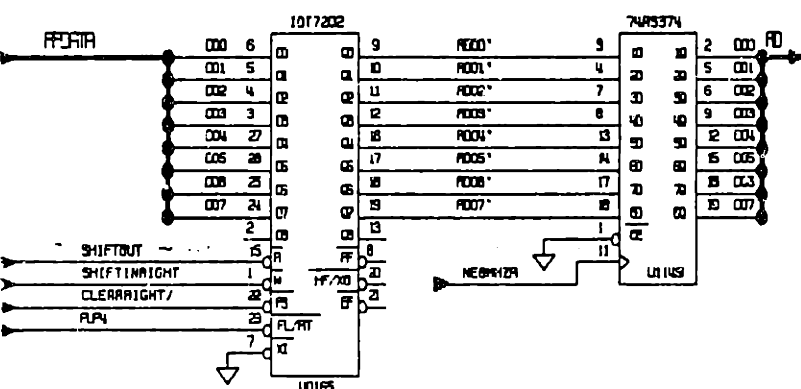
A.5 Sheet 5

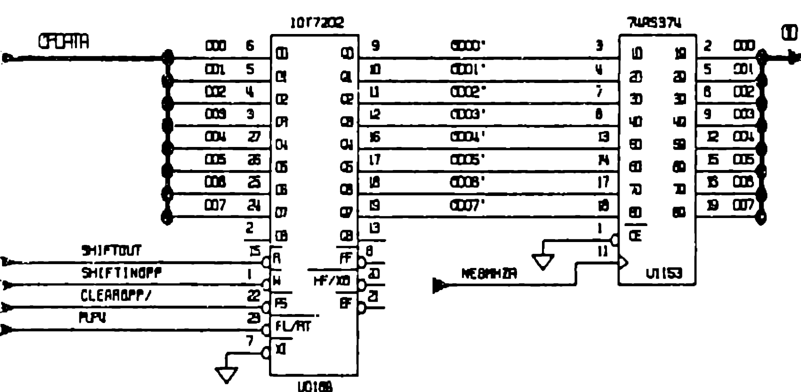






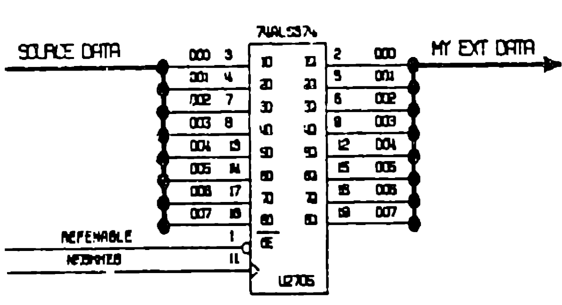


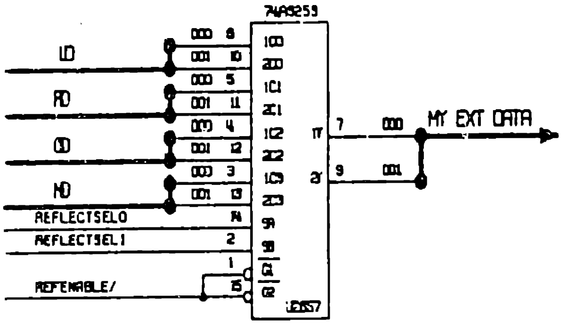


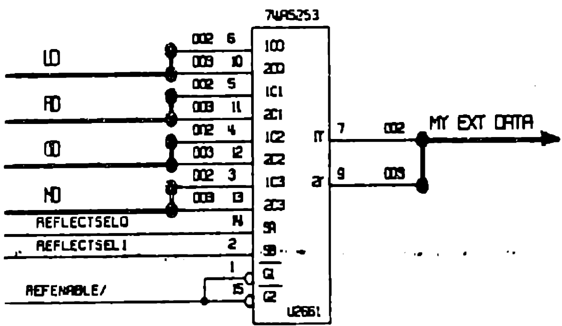


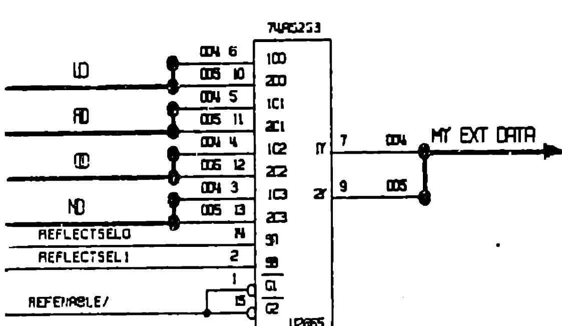
FILES: VOTER			
TITLE: DATA PATH FIFOS, VOTER			
SIZE: B	CODE	NUMBER	REV: C
DATE: 04/17/88		SHEET: 5 OF 12	

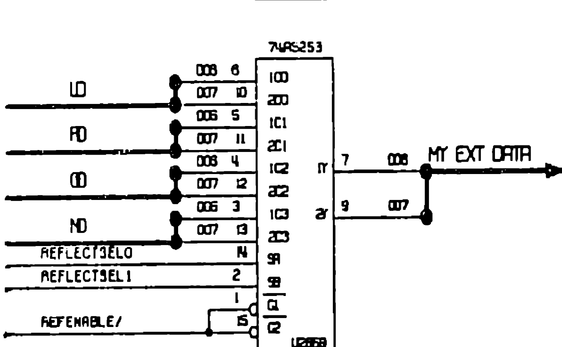
A.6 Sheet 6

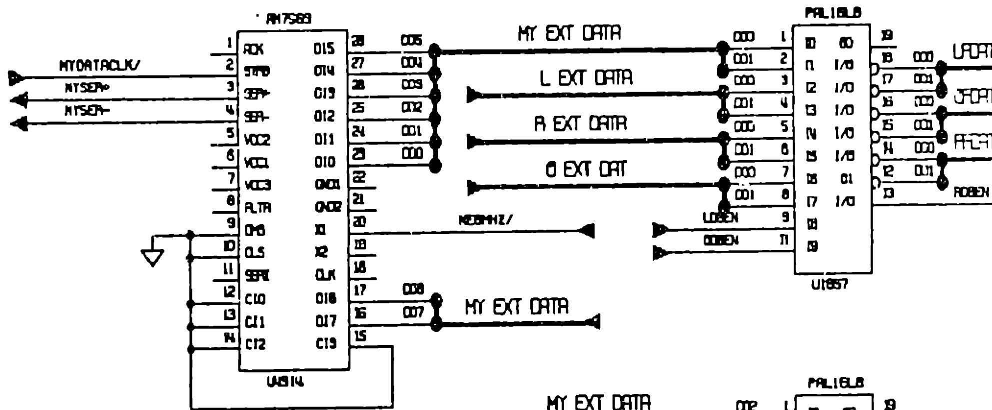


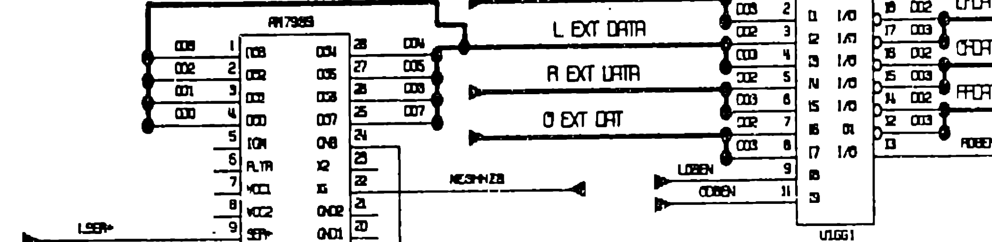


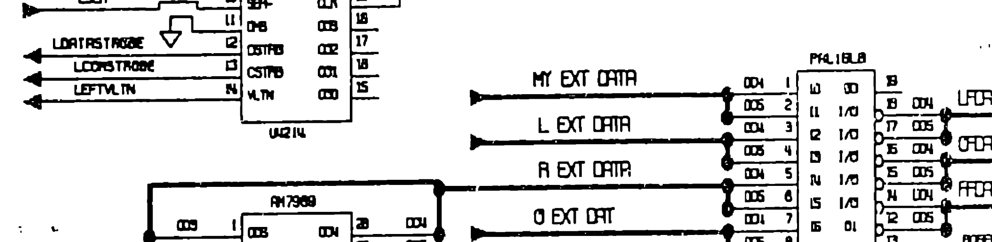


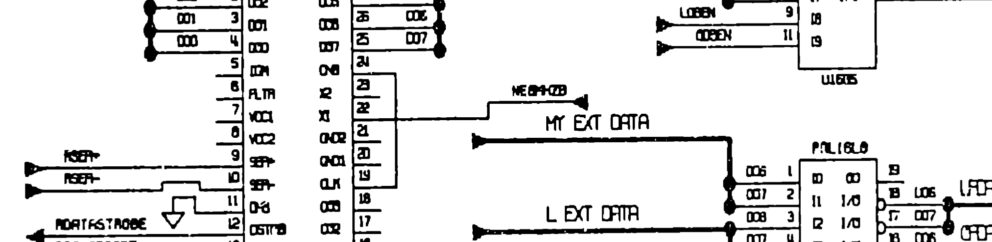


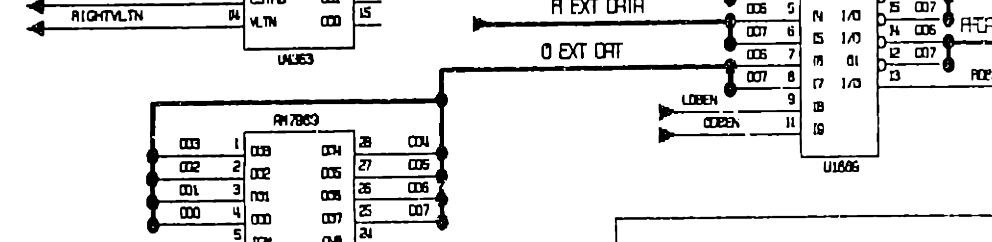


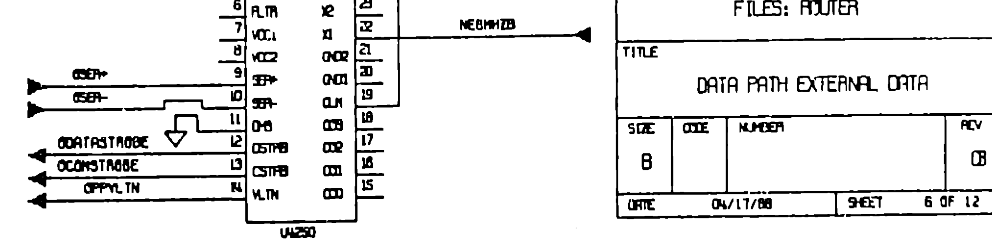




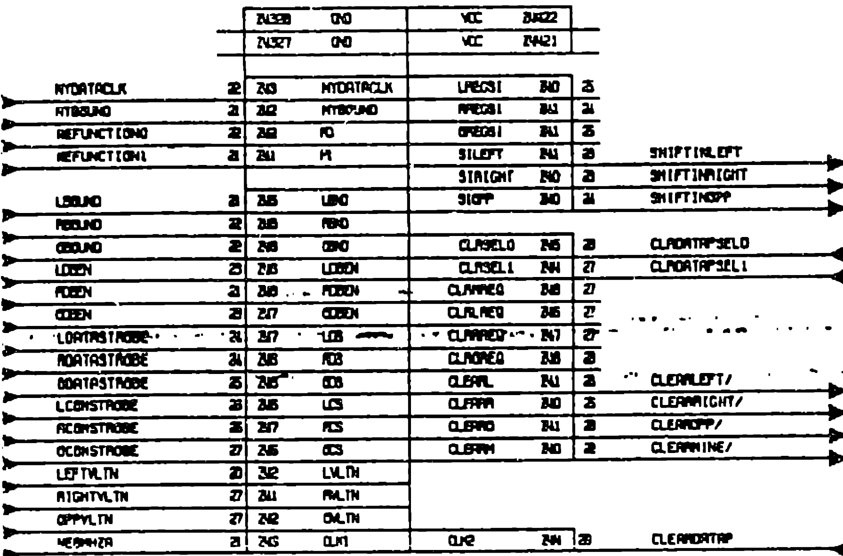


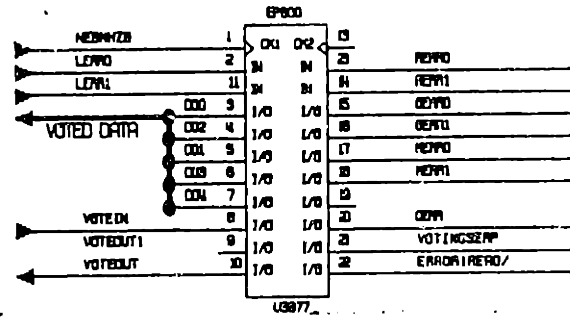


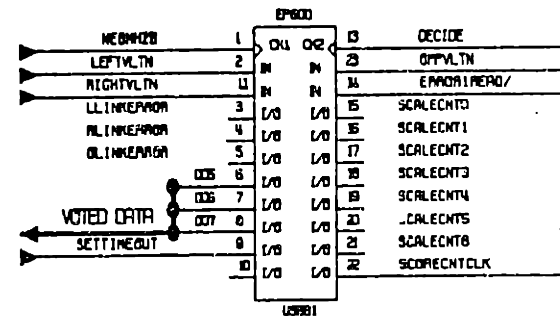


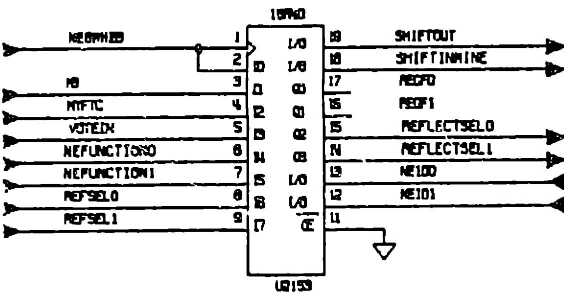


A.7 Sheet 7









FILES: SYNCDATACONT, ASYNCDATACONT SYNACC, LINKERRACC			
TITLE DATA PATH CONTROL AND ERROR ACCUM			
SIZE B	CODE	NUMBER	REV 03
DATE 04/17/88		SHEET 7 OF 12	

A.8 Sheet 8

SCORE BOARD 1ST HALF EP900

SCORE BOARD 2ND HALF EP900

FILES: SCORE1, SCORE2

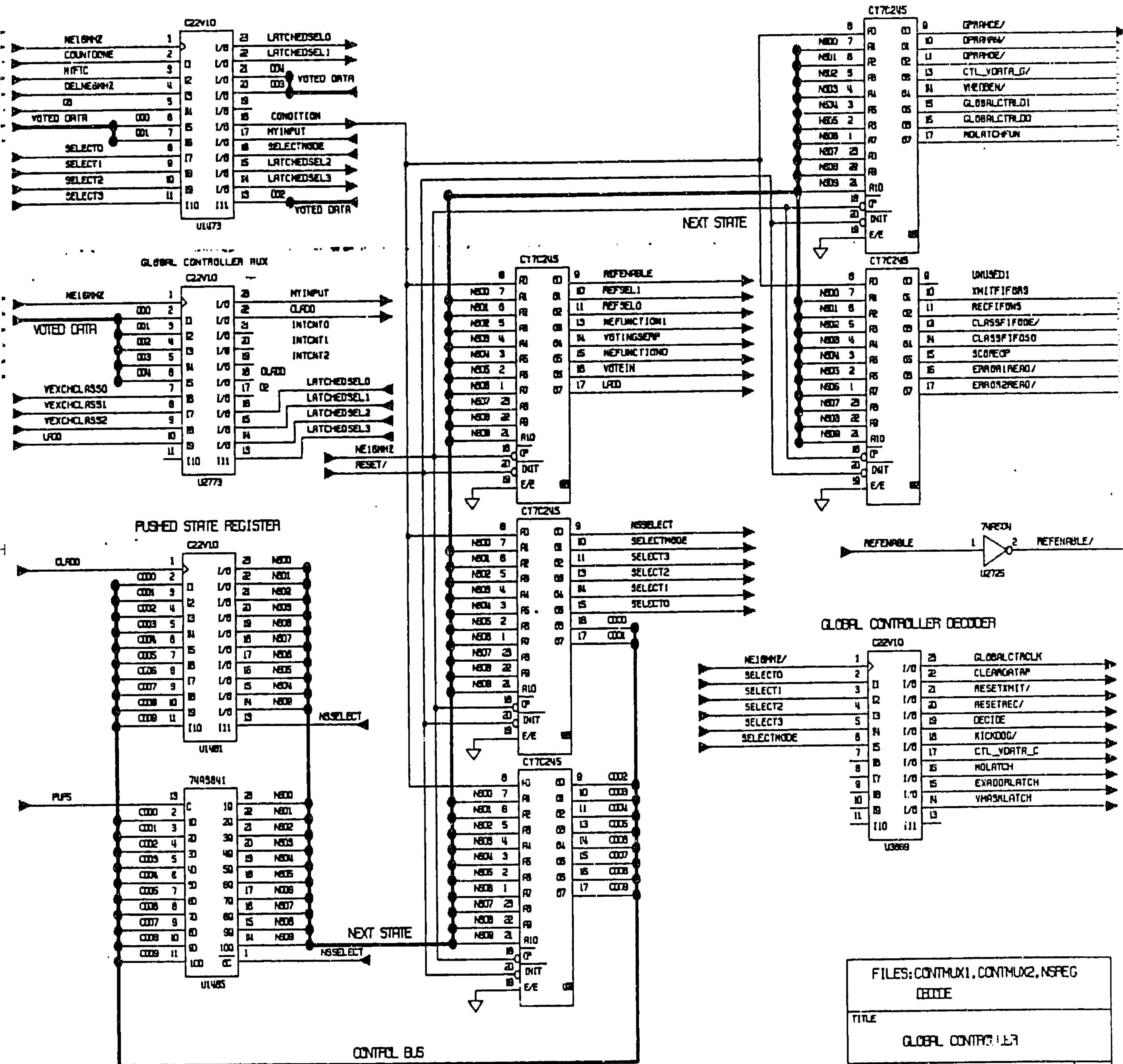
TITLE

SCORE BOARD

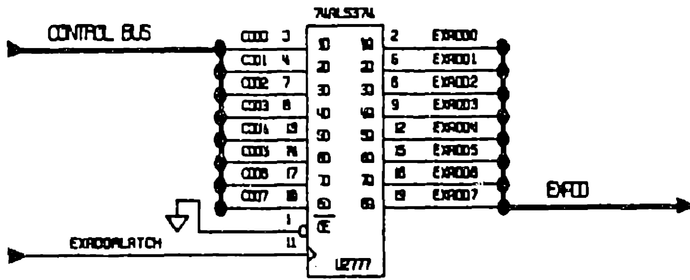
SIZE	CODE	NUMBER	REV
B			08

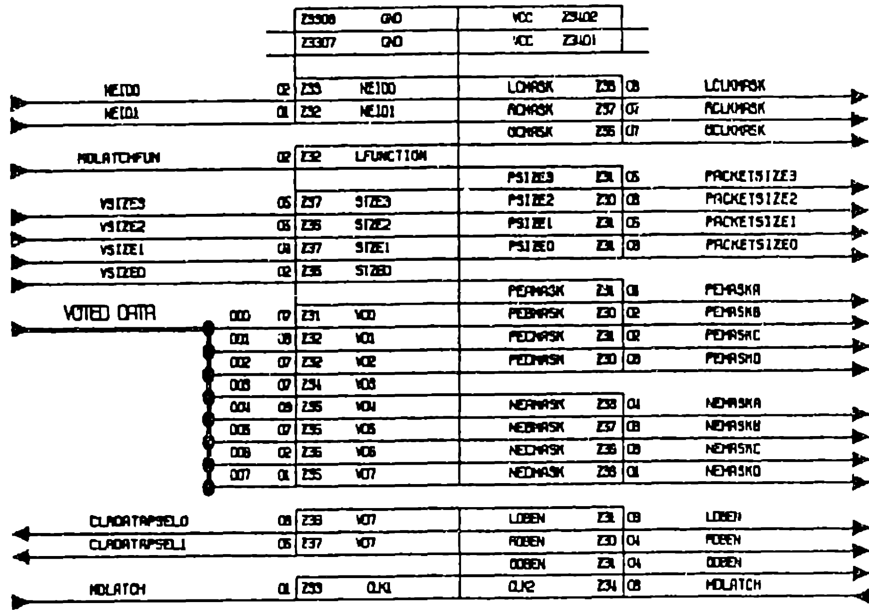
DATE 04/17/88 SHEET 6 OF 12

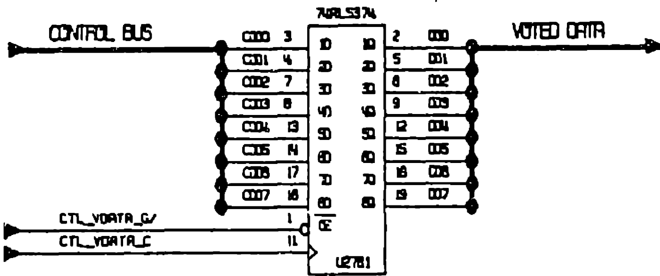
A.9 Sheet 9

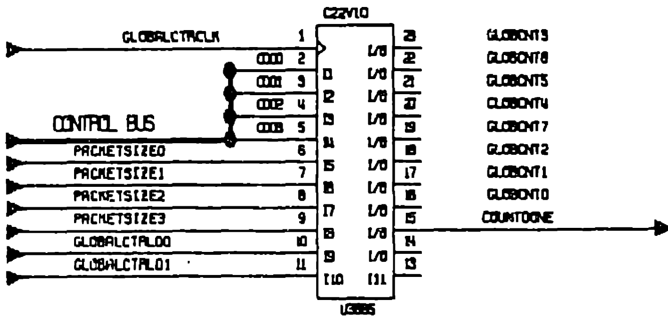


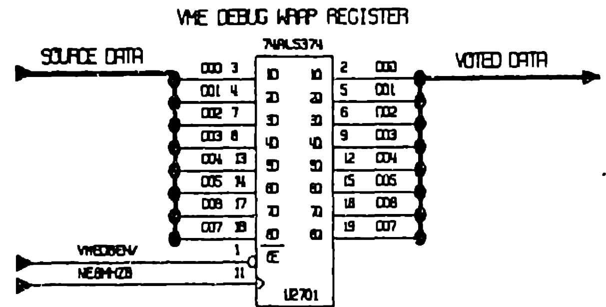
A.10 Sheet 10

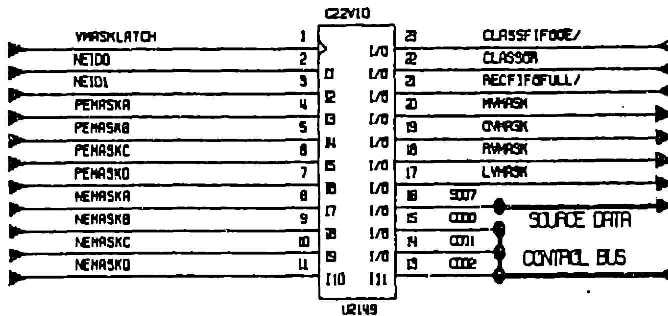










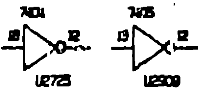


FILES: MASKREG, VMASKREG, GLOBCNT			
TITLE GLOBAL CONTROLLER 2			
SIZE B	CODE	NUMBER	REV C8
DATE 04/17/88		SHEET 10 OF 12	

A.11 Sheet 11

NE ID SWITCH

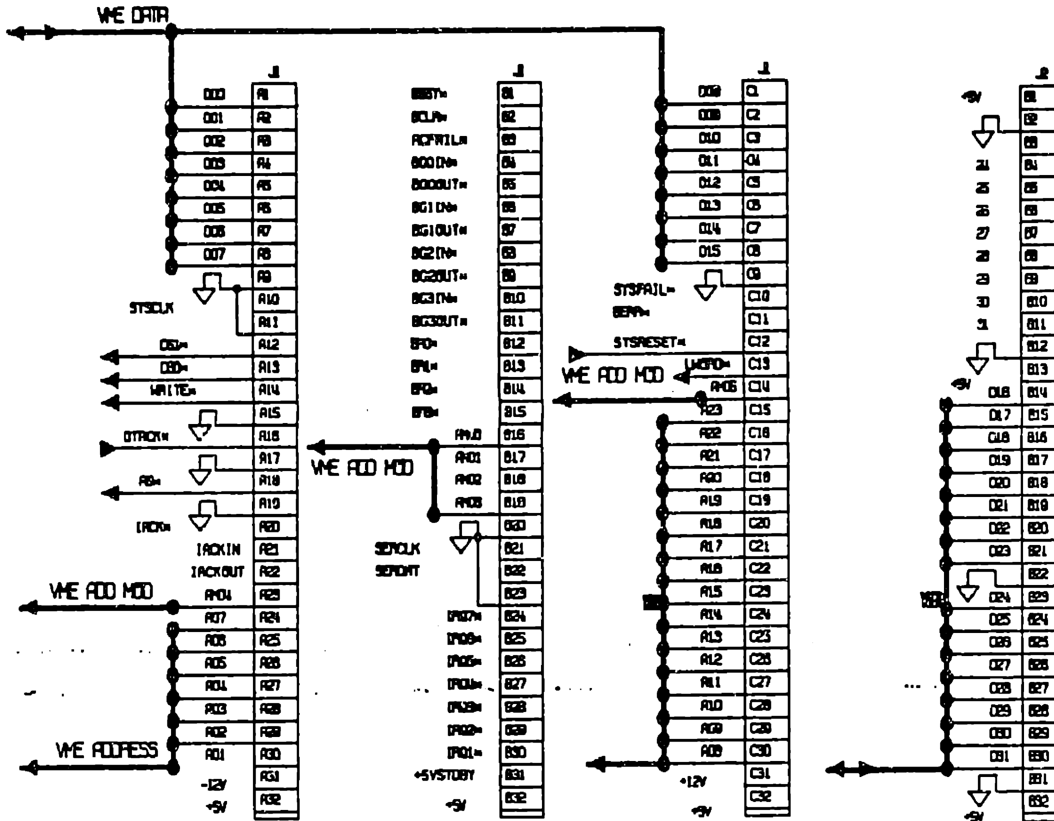
SPARES



FILES:

TITLE	FIBER OPTIC INTERFACE RESET PUPS AND SPARES		
SIZE	CODE	NUMBER	REV
B			03
DATE	04/17/88	SHEET	11 OF 12

A.12 Sheet 12



FILE:			
TITLE	BUS INTERFACE		
SIZE B	CODE	NUMBER	REV 03
DATE	04/17/88	SHEET	12 OF 12

A.13 Sheet 13

Appendix B
PLD Equations

B.1 VME Interface

B.1.1 Address Decoding

```
MODULE ADDDECODE
TITLE 'VME ADDRESS DECODER, 22 OCT 87'

DECODE DEVICE 'E0310';
AM00,AM03,AM04,AM05 PIN 2,3,4,5;
A23,A22,A21,A20,A19,A18,A17 PIN 6,7,8,9,11,12,13;
A16,A15,A14 PIN 14,15,16;
!LWORD PIN 17;

!BOARDSEL PIN 18;

AMOD = [AM05,AM04,AM03,AM00];
ADD = [A23,A22,A21,A20,A19,A18,A17,A16,A15,A14];
VALID = [0,0,0,0,0,0,0,0,0,0];

A18,A17,A16,A15,A14 ISTYPE 'feed_pin';
!LWORD ISTYPE 'FEED_PIN';
!BOARDSEL ISTYPE 'NEG,COM';

EQUATIONS

BOARDSEL = (ADD == VALID) & (AMOD == ^HF) & LWORD;

END ADDDECODE
```

B.1.2 VME Block Transfer Address Counter

```
MODULE VMECOUNTER FLAG '-R3'
TITLE 'VME BLOCK XFER ADDRESS COUNTER, 4 DEC 87'
```

```
VMECNT DEVICE 'P16R6';

CTRCLK PIN 1;
!CTRLOAD,GND PIN 2,11;
A02,A03,A04,A05,A06,A07 PIN 3,4,5,6,7,8;
PA02,PA03,PA04,PA05,PA06,PA07 PIN 13,14,15,16,17,18;

COUNT = [PA07,PA06,PA05,PA04,PA03,PA02];
INPUT = [A07,A06,A05,A04,A03,A02];

EQUATIONS

COUNT := (COUNT + 1) & !CTRLOAD
# INPUT & CTRLOAD;

END VMECOUNTER
```

B.1.3 Data Strobe Generator

```
MODULE DSGEN
TITLE 'VME DATA STROBE GENEREATOR, 22 OCT 87'

DSGN DEVICE 'E0600';
!BSEL,!AS,!DS0,!DS1,!WRITE PIN 2,3,4,5,6;
A13,A12,A11 PIN 14,11,10;
RXMITFIFODS,RECFIFODSH,CLASSFIFODS PIN 21,20,19;
CAB,!STATUSREGDSL,!DPRAMDS PIN 18,17,16;
!CTLOAD,CTRCLK PIN 7,9;
!DPRAMWRITE,WRDONE PIN 22,23;
!BOARDG PIN 15;

!CTLOAD ISTYPE 'NEG,REG_D,FEED_REG';

GO = [BSEL,AS,DS0,DS1];
SELECT = [A13,A12,A11];
RXDS,RDS,CDS = RXMITFIFODS,RECFIFODSH,CLASSFIFODS;
SDL,DDS = STATUSREGDSL,DPRAMDS;
CL,CC = CTLOAD,CTRCLK;
WD = WRDONE;
X = .X.;
```

```
EQUATIONS

RXMITFIFODS := 1;
RXMITFIFODS.C = (GO == 15) & (SELECT == 3) & WRITE;
RXMITFIFODS.RE = !WRDONE;

RECFIFODSH = (GO == 15) & (SELECT == 1) & !WRITE;

CLASSFIFODS = (GO == 15) & (SELECT == 2) & WRITE;

STATUSREGDSL = (GO == 15) & (SELECT == 0) & !WRITE;

DPRAMDS = (GO ==15) & A13;

DPRAMWRITE = (GO == 15) & A13 & WRITE;

CAB = DS0 & DS1 & WRITE;

BOARDG = (GO == 15);

CTLOAD := 0;
CTLOAD.C = CTRCLK;
CTLOAD.RE = !AS;

CTRCLK = (AS & DS0);


FUSES
6461 = 1;
6406 = 1;

END DSGEN
```

B.1.4 DTACK Generator

```
MODULE DTACKGEN
TITLE 'VME DTACK GENERATOR PAL, 20 OCT 87'

DTCK DEVICE 'E0600';
NE16MHZ PIN 1;
```

```
!STATUSREGDS,!DPRAMDS,CLASSFIFODS PIN 2,11,14;
XMITFIFODS,RECFIFODS,RDDONE,WRDONE PIN 23,22,21,20;
SCRDTACKA,SCRDTACK PIN 3,4;
XMITFIFODTACK,RECFIFODTACK PIN 5,6;
RRECFIFODS,DTACK PIN 18,7;
!BUSY,NE8MHZ,DELNE8MHZ PIN 9,10,8;
RESETPE,RESETNE,RESETPROC PIN 17,16,15;
CLASSSI PIN 19;

X = .X.;
XDT,RDT = XMITFIFODTACK,RECFIFODTACK;
RRDS = RRECFIFODS;
XDS,RDS = XMITFIFODS,RECFIFODS;

FUSES
6451 = 1;
6456 = 1;
6421 = 1;

EQUATIONS

SCRDTACKA := STATUSREGDS # CLASSFIFODS # (DPRAMDS & !BUSY);

SCRDTACK := (SCRDTACKA & !BUSY) # (SCRDTACK & DPRAMDS);

XMITFIFODTACK := 1;
XMITFIFODTACK.C = WRDONE;
XMITFIFODTACK.RE = !XMITFIFODS;

RECFIFODTACK := 1;
RECFIFODTACK.C = RDDONE;
RECFIFODTACK.RE = !RECFIFODS;

RRECFIFODS := 1;
RRECFIFODS.C = RECFIFODS;
RRECFIFODS.RE = !RDDONE;

DTACK = SCRDTACK # XMITFIFODTACK # RECFIFODTACK;

RESETPROC = RESETPE # RESETNE;

DELNE8MHZ = NE8MHZ;
```

```
CLASSSI = SCRDTACK & CLASSFIFODS;

END DTACKGEN
```

B.1.5 VME FIFO Control 1

```
MODULE FIFOCONT1 FLAG '-R3'
TITLE 'VME FIFO CONTROLLER PAL 1ST HALF, 19 OCT 87'

FIFO1 DEVICE 'P20R6';
NE8MHZ_BAR,NE8MHZA_BAR,RDDONE,!RESETREC PIN 1,6,8,9;
RXMITFIFODS,RRECFIFODS,!RESETXMIT,WRDONE,!CBA3 PIN 2,3,4,5,7;
!G0,!G1,!G2,!G3 PIN 20,19,18,17;
!BOARDSEL,RECFIFODS,STARTRECSHIFT,STARTXMITSHIFT,GND
              PIN 10,11,21,16,13;
XMITFIFOWS PIN 22;

N8,RXDS,RRDS = !NE8MHZ_BAR,RXMITFIFODS,RRECFIFODS;
RX,RR,WRD,RD = RESETXMIT,RESETREC,WRDONE,RDDONE;
SXS,SRS = STARTXMITSHIFT,STARTRECSHIFT;
N8A,WS,RDS = !NE8MHZA_BAR,XMITFIFOWS,RECFIFODS;
BS = BOARDSEL;
C,X = .C.,.X.;
EQUATIONS

STARTXMITSHIFT := (RXMITFIFODS & !G2 & WRDONE)
# (STARTXMITSHIFT & !G2 & !RESETXMIT);

STARTRECSHIFT := (RRECFIFODS & RDDONE)
# (STARTRECSHIFT & !CBA3 & !RESETREC);

G0 := STARTXMITSHIFT & !G0 & !G1 & !G2  & !RESETXMIT
    # RECFIFODS & !RESETREC;
G1 := G0 & !WRDONE & !RESETXMIT # (RECFIFODS & !RESETREC);
G2 := G1 & !WRDONE & !RESETXMIT # (RECFIFODS & !RESETREC);
G3 := G2 & !WRDONE & !RESETXMIT # (RECFIFODS & !RESETREC)
# (BOARDSEL & !STARTXMITSHIFT);

XMITFIFOWS = !NE8MHZA_BAR # WRDONE # RESETXMIT;
```

```
END FIFOCONT1
```

B.1.6 VME FIFO Control 2

```
MODULE FIFOCONT2
TITLE 'VME FIFO CONTROLLER PAL 2ND HALF, 19 OCT 87'

FIFO2 DEVICE 'P16R6';
READCLK,READCLKO,RECFIFORS,GND PIN 1,19,12,11;
STARTRECSHIFT,!RESETREC,VME16MHZ_BAR,STARTXMITSHIFT PIN 2,3,4,5;
!RECFIFOEMPTY PIN 7;
!CBA0,!CBA1,!CBA2,!CBA3,RDDONE,WRDONE PIN 18,17,16,15,14,13;

C,X = .C.,.X.;
SRS = STARTRECSHIFT;
E = RECFIFOEMPTY;
RCLK,VCLK = READCLK,!VME16MHZ_BAR;
RREC,RRS,RCLKO = RESETREC,RECFIFORS,READCLKO;

EQUATIONS

WRDONE := !STARTXMITSHIFT;
RDDONE := !STARTRECSHIFT;

CBA0 := STARTRECSHIFT & !CBA0 & !CBA1 & !CBA2 & !CBA3 & !RESETREC;
CBA1 := CBA0 & !RESETREC;
CBA2 := CBA1 & !RESETREC;
CBA3 := CBA2 & !RESETREC;

RECFIFORS = READCLKO # RDDONE # RESETREC;

READCLKO = !VME16MHZ_BAR & (!RECFIFOEMPTY # RDDONE);

END FIFOCONT2
```

B.2 Fault Tolerant Clock

B.2.1 MYFTC Generator

```
MODULE MYFTCGEN
TITLE 'MYFTC GENERATOR PAL, 29 OCT 87'

MYFTCGN DEVICE 'E0900';
NE8MHZ,NE8MHZ2 PIN 1,21;
ST3,ST2,ST1,ST0 PIN 5,6,7,8;
EW,SAW,SBW,MYFTC,LD PIN 9,10,11,12,13;
COUNT_119,COUNT_116,COUNT_127 PIN 14,15,16;
!RI,SB,SB1,SA,SA1 PIN 25,26,27,28,29;
FTCLATCH,SIZE3,SIZE2,SIZE1,SIZE0 PIN 38,39,2,3,4;
C6,C5,C4,C3,C2,C1,C0 PIN 36,35,34,33,32,31,30;
MEDBND,!RE PIN 17,18;

SB,SB1,SA,SA1 ISTYPE 'FEED_REG';
COUNT_119,COUNT_116,COUNT_127 ISTYPE 'FEED_REG';

CURRENT_STATE = [ST3,ST2,ST1,ST0];
S0,S1,S2,S3 = ^B0000,^B0010,^B1110,^B1111;
S4,S5,S6,S7 = ^B1101,^B1100,^B1010,^B1011;
S8,S9,S10,S11 = ^B1001,^B0001,^B0011,^B0110;
S12,S13,S14,S15 = ^B0101,^B0111,^B1000,^B0100;
OUTPUTS = [EW,SAW,SBW,MYFTC];
COUNT = [C6,C5,C4,C3,C2,C1,C0];
SIZE = [SIZE3,SIZE2,SIZE1,SIZE0];
LOAD = [LD,LD,LD,LD,LD,LD,LD];
RESETI = [RI,RI,RI,RI,RI,RI,RI];
RESET = [RE,RE,RE,RE,RE,RE,RE];
CK1,CK2 = NE8MHZ,NE8MHZ2;
C = .C.;
X = .X.;


STATE_DIAGRAM CURRENT_STATE

STATE S0: IF (!COUNT_119)
 THEN S0
WITH OUTPUTS:=^B1010;
ENDWITH
 ELSE S1
WITH OUTPUTS:=^B1010;
ENDWITH;
STATE S1: IF (SIZE == 0)
```

```
THEN S1
WITH OUTPUTS:=^B1010;
ENDWITH
 ELSE IF !SA
THEN S3
WITH OUTPUTS:=^B1011;
ENDWITH
 ELSE  S2
WITH OUTPUTS:=^B1010;
ENDWITH;
STATE S2: GOTO S3
WITH OUTPUTS:=^B1011;
ENDWITH;
STATE S3: GOTO S4
WITH OUTPUTS:=^B1011;
ENDWITH;
STATE S4: GOTO S5
WITH OUTPUTS:=^B1011;
ENDWITH;
STATE S5: GOTO S6
WITH OUTPUTS:=^B1011;
ENDWITH;
STATE S6: GOTO S7
WITH OUTPUTS:=^B0011;
ENDWITH;
STATE S7: GOTO S8
WITH OUTPUTS:=^B0001;
ENDWITH;
STATE S8: GOTO S9
WITH OUTPUTS:=^B0101;
ENDWITH;
STATE S9: GOTO S10
WITH OUTPUTS:=^B1101;
ENDWITH;
STATE S10: IF (!COUNT_127)
THEN S10
WITH OUTPUTS:=^B1101;
ENDWITH
 ELSE IF (SB1)
THEN S12
WITH OUTPUTS:=^B1101;
ENDWITH
```

```
 ELSE S11
WITH OUTPUTS:=^B1101;
ENDWITH;
STATE S11: GOTO S12
WITH OUTPUTS:=^B1101;
ENDWITH;
STATE S12: GOTO S13
WITH OUTPUTS:=^B1101;
ENDWITH;
STATE S13: IF (!COUNT_116)
THEN S14
WITH OUTPUTS:=^B1111;
ENDWITH
  ELSE IF (SB)
THEN S0
WITH OUTPUTS:=^B1111;
ENDWITH
  ELSE S0
WITH OUTPUTS:=^B1110;
ENDWITH;
STATE S14: IF (!COUNT_116)
THEN S14
WITH OUTPUTS:=^B1011;
ENDWITH
  ELSE IF (SB)
THEN S0
WITH OUTPUTS:=^B1011;
ENDWITH
  ELSE S0
WITH OUTPUTS:=^B1010;
ENDWITH;
STATE S15: GOTO S0;

EQUATIONS

LD := !ST3 & !ST2 & ST1 & !ST0 & !(SIZE == 0) & !SA
# ST3 & ST2 & ST1 & !ST0
# !ST3 & !ST2 & ST1 & ST0 & COUNT_127 & SB1
# !ST3 & ST2 & ST1 & !ST0;

COUNT := (COUNT + 1) & !LOAD
# [!SIZE3,!SIZE2,!SIZE1,!SIZE0,1,0,0] & LOAD;
```

```
C6.RE = RE;
C5.RE = RE;
C4.RE = RE;
C3.RE = RE;
C2.RE = RE;
C1.RE = RE;
C0.RE = RE;

COUNT_119 := (COUNT == 118);

COUNT_127 := (COUNT == 126);

COUNT_116 := (COUNT == 115) # (LD & (SIZE == 1));

RI = (SIZE == 0);

SA1 := SAW & !SBW;
SA1.C = MEDBND;
SA1.RE = RE;

SA := SA1 & !ST3 & !ST2 & ST1 & ST0
# SA & !(!ST3 & !ST2 & ST1 & ST0);
SA.RE = RE;

SB1 := SBW;
SB1.C := MEDBND;
SB1.RE = RE;

SB := SB1 & !ST3 & !ST2 & ST1 & ST0
# SB & !(!ST3 & !ST2 & ST1 & ST0);
SB.RE = RE;

ST3.RE = RE;
ST2.RE = RE;
ST1.RE = RE;
ST0.RE = RE;

FUSES
17316 = 1;
17326 = 1;

END MYFTCGEN
```

B.2.2 Median Bound Voter and Clock Error Accumulator

```
MODULE FTC2
TITLE 'BOUND VOTER AND ERROR ACCUMULATION, 4 NOV 87'

FTC DEVICE 'E0900';

LBOUND,RBOUND,OBOUND,MEDBOUND PIN 2,3,4,5;
MYFTC,HOLDMEDBND,PRESWINDOW PIN 6,7,14;
!XFULL,!CLASSFULL,CTS PIN 8,9,10;
LMASK,OMASK,RMASK PIN 17,18,19;
DELMEDBND,EW,!ERR2READ PIN 22,23,24;
MACCERR,MERR PIN 36,35;
DLB,DOB,DRB PIN 39,38,37;
LBERR,LAERR,LPRES,LACCERR PIN 34,33,32,31;
RBERR,RAERR,RPRES,RACCERR PIN 30,29,28,27;
OBERR,OAERR,OPRES,OACCERR PIN 26,25,16,15;
NE8MHZ,NE8MHZA PIN 1,21;
LERR,OERR,RERR PIN 11,12,13;

ERROR = [MACCERR,LACCERR,RACCERR,OACCERR];

LBERR,LAERR,LPRES ISTYPE 'POS,REG,FEED_REG';
RBERR,RAERR,RPRES ISTYPE 'POS,REG,FEED_REG';
OBERR,OAERR,OPRES ISTYPE 'POS,REG,FEED_REG';
MERR,LACCERR,RACCERR,OACCERR,MACCERR ISTYPE 'POS,REG,FEED_REG';
DLB,DOB,DRB,LBOUND,RBOUND,OBOUND ISTYPE 'FEED_PIN';
  DELMEDBND,MEDBOUND,MYFTC,HOLDMEDBND ISTYPE 'FEED_PIN';
PRESWINDOW ISTYPE ',POS,REG,FEED_REG';
XFULL,CLASSFULL,CTS ISTYPE 'FEED_PIN';

FUSES
"FOR LERR,RERR,OERR,MERR ACCUMULATED
17286 = 1;
17291 = 1;
17296 = 1;
17301 = 1;
17311 = 1;
17316 = 1;
17321 = 1;
```

```
17331 = 1;
17336 = 1;
17396 = 1;
17386 = 1;

EQUATIONS

MEDBOUND = LBOUND & LMASK & RBOUND & RMASK
  #LBOUND & LMASK & OBOUND & OMASK
  #RBOUND & RMASK & OBOUND & OMASK
  #LBOUND & LMASK & !RMASK & !OMASK
  #RBOUND & RMASK & !LMASK & !OMASK
  #OBOUND & OMASK & !LMASK & !RMASK
  # HOLDMEDBND;

HOLDMEDBND = MEDBOUND & LMASK & OMASK & !RMASK & !(!LBOUND & !OBOUND)
# MEDBOUND & LMASK & RMASK & !OMASK & !(!LBOUND & !RBOUND)
# MEDBOUND & RMASK & OMASK & !LMASK & !(!RBOUND & !OBOUND);

LBERR := !LBOUND;
LBERR.C = DELMEDBND;
LBERR.RE = ERR2READ;

LAERR := !MEDBOUND;
LAERR.C = DLB;
LAERR.RE = ERR2READ;

LPRES := 1;
LPRES.C = LBOUND;
LPRES.RE = !MYFTC;

LACCERR := (LACCERR # (!LPRES & PRESWINDOW) # LBERR # LAERR) & !ERR2READ;

RBERR := !RBOUND;
RBERR.C = DELMEDBND;
RBERR.RE = ERR2READ;

RAERR := !MEDBOUND;
RAERR.C = DRB;
RAERR.RE = ERR2READ;

RPRES := 1;
```

```
RPRES.C = RBOUND;
RPRES.RE = !MYFTC;

RACCERR := (RACCERR # (!RPRES & PRESWINDOW) # RBERR # RAERR) & !ERR2READ;

OBERR := !OBOUND;
OBERR.C = DELMEDBND;
OBERR.RE = ERR2READ;

OAERR := !MEDBOUND;
OAERR.C = DOB;
OAERR.RE = ERR2READ;

OPRES := 1;
OPRES.C = OBOUND;
OPRES.RE = !MYFTC;

OACCERR := (OACCERR # (!OPRES & PRESWINDOW) # OBERR # OAERR) & !ERR2READ;

PRESWINDOW := 1;
PRESWINDOW.C = DELMEDBND;
PRESWINDOW.RE = !MYFTC;

MERR := EW;
MERR.C = MEDBOUND;

MACCERR := (MACCERR # MERR) & !ERR2READ;

ENABLE ERROR = ERR2READ;

CTS = !XFULL & !CLASSFULL;

LERR = LACCERR;

RERR = RACCERR;

OERR = OACCERR;

END FTC2
```

B.2.3 Bound Generator

```
MODULE BOUNDGEN
TITLE 'EXTERNAL BOUND GENERATOR PAL, 5 NOV 87'

BNDGEN DEVICE 'P20RA10';

LDS,RDS,ODS,LCS,RCS,OCS PIN 2,3,4,5,6,7;
LVLTN,RVLTN,OVLTN PIN 8,9,10;
GND,PUP PIN 13,1;
LBOUND,RBOUND,OBOUND PIN 23,22,21;
LBOUND1,RBOUND1,OBOUND1 PIN 20,19,18;
DELLBOUND,DELRBOUND,DELOBOUND PIN 17,16,15;

DS = [LDS,RDS,ODS];
CS = [LCS,RCS,OCS];
BOUND = [LBOUND,RBOUND,OBOUND];
VLTN = [LVLTN,RVLTN,OVLTN];
X = .X.;

EQUATIONS


LBOUND := 1;
LBOUND.C = LDS;
LBOUND.PR = LCS & !LVLTN;
LBOUND.RE = LCS & LVLTN;

LBOUND1 := LBOUND;
LBOUND1.C = LDS;
LBOUND1.PR = LCS & !LVLTN;

DELLBOUND := LBOUND1;
DELLBOUND.C = !LDS;
DELLBOUND.PR = LCS & !LVLTN;

RBOUND := 1;
RBOUND.C = RDS;
RBOUND.PR = RCS & !RVLTN;
RBOUND.RE = RCS & RVLTN;

RBOUND1 := RBOUND;
```

```
RBOUND1.C = RDS;
RBOUND1.PR = RCS & !RVLTN;

DELRBOUND := RBOUND1;
DELRBOUND.C = !RDS;
DELRBOUND.PR = RCS & !RVLTN;

OBOUND := 1;
OBOUND.C = ODS;
OBOUND.PR = OCS & !OVLTN;
OBOUND.RE = OCS & OVLTN;

OBOUND1 := OBOUND;
OBOUND1.C = ODS;
OBOUND1.PR = OCS & !OVLTN;

DELOBOUND := OBOUND1;
DELOBOUND.C = !ODS;
DELOBOUND.PR = OCS & !OVLTN;

END BOUNDGEN
```

B.2.4 System Clock Generator

```
MODULE CLOCKGEN FLAG '-R3'
TITLE 'SYSTEM CLOCK GENERATOR PAL, 11 NOV 87'

CLKGEN DEVICE 'P16R4';

C32MHZ,GND PIN 1,11;
MYFTC,!RESET PIN 2,3;
C16MHZ,C8MHZ,MB1,MB PIN 17,16,14,15;
AQERR,AQERR0,AQERR1,BQERR,BQERR0,BQERR1 PIN 4,5,6,7,8,9;
MYDATACLK,!MYDATCLK PIN 13,12;
QERR PIN 19;


COUNT = [!C8MHZ,!C16MHZ];
MDC,MC = MYDATACLK,MYDATCLK;
C32 = C32MHZ;
C = .C.;
```

```
EQUATIONS

COUNT := (COUNT + 1);

MYDATACLK = MB & C8MHZ;

MYDATCLK = MB & C8MHZ;

MB := MYFTC & !C8MHZ & !C16MHZ & !RESET
 # MB & (C8MHZ # C16MHZ) & !RESET;

MB1 := MB;

QERR = AQERR # AQERR0 # AQERR1 # BQERR # BQERR0 # BQERR1;

END CLOCKGEN
```

B.3 Data Paths

B.3.1 Data Path Voter Slice

```
MODULE VOTER
"MODULE VOTER FLAG '-R3'"
TITLE 'VOTER PAL, 30 DEC 87'

VOTE DEVICE 'E0900';
LVMASK,OVMASK,RVMASK,MVMASK PIN 2,3,4,39;
LD0,LD1,LD2,LD3,OD0,OD1,OD2,OD3 PIN 5,6,7,8,9,10,11,12;
RD0,RD1,RD2,RD3,MD0,MD1,MD2,MD3 PIN 16,17,18,19,22,23,24,37;
VD0,VD1,VD2,VD3 PIN  36,35,34,33;
LERR,RERR,MERR,OERR,QERR0,QERR1 PIN 32,31,30,29,28,27;
VOTEOUT,QERR PIN 38,13;
CVD0,CVD1,CVD2,CVD3 PIN 25,26,15,14;
NE8MHZ PIN 21;

LEFT = [LD3,LD2,LD1,LD0];
RIGHT = [RD3,RD2,RD1,RD0];
  OPP = [OD3,OD2,OD1,OD0];
MINE = [MD3,MD2,MD1,MD0];
```

```
VOTED = [VD3,VD2,VD1,VD0];
CVOTED = [CVD3,CVD2,CVD1,CVD0];
MASK = [LVMASK,RVMASK,OVMASK,MVMASK];
MASKL = [LVMASK, LVMASK, LVMASK, LVMASK];
MASKR = [RVMASK, RVMASK, RVMASK, RVMASK];
MASKO = [OVMASK, OVMASK, OVMASK, OVMASK];
MASKM = [MVMASK, MVMASK, MVMASK, MVMASK];

ERRORS = [LERR,RERR,OERR,MERR];
QUADS = [QERR0, QERR1, QERR];
VO = VOTEOUT;
CVD0,CVD1,CVD2,CVD3 ISTYPE 'POS,COM,FEED_PIN';
VD0, VD1, VD2, VD3 istype 'feed_REG';
LERR,RERR,MERR,OERR istype 'com';
QERR0,QERR1 istype 'COM,feed_pin';
QERR istype 'COM,feed_pin';

EQUATIONS

CVOTED = (LEFT & MASKL & RIGHT & MASKR)
    # (LEFT & MASKL & MINE & MASKM)
      # (LEFT & MASKL & OPP & MASKO)
    # (RIGHT & MASKR & MINE & MASKM)
    # (RIGHT & MASKR & OPP & MASKO)
    # (OPP & MASKO & MINE & MASKM)
    # (MINE & MASKM & !MASKL & !MASKO & !MASKR)
    # (OPP & MASKO & !MASKM & !MASKL & !MASKR);

VOTED := CVOTED;

ENABLE VOTED = VOTEOUT;
ENABLE ERRORS = 15;
ENABLE QUADS = 7;
ENABLE CVOTED = 15;

QERR = (
(LD0 & RD0 & !MD0 & !OD0)
    # (!LD0 & !RD0 & MD0 & OD0)
    # (LD0 & !RD0 & MD0 & !OD0)
    # (!LD0 & RD0 & !MD0 & OD0)
    # (LD0 & !RD0 & !MD0 & OD0)
    # (!LD0 & RD0 & MD0 & !OD0)
```

```
    # (LD3 & RD3 & !MD3 & !OD3)
    # (!LD3 & !RD3 & MD3 & OD3)
)
 & (LVMASK & RVMASK & OVMASK & MVMASK);

QERR0 = (LD1 & RD1 & !MD1 & !OD1 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD1 & !RD1 & MD1 & OD1 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (LD1 & !RD1 & MD1 & !OD1 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD1 & RD1 & !MD1 & OD1 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (LD1 & !RD1 & !MD1 & OD1 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD1 & RD1 & MD1 & !OD1 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (LD3 & !RD3 & MD3 & !OD3 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD3 & RD3 & !MD3 & OD3 & LVMASK & RVMASK & OVMASK & MVMASK);

QERR1 = (LD2 & RD2 & !MD2 & !OD2 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD2 & !RD2 & MD2 & OD2 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (LD2 & !RD2 & MD2 & !OD2 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD2 & RD2 & !MD2 & OD2 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (LD2 & !RD2 & !MD2 & OD2 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD2 & RD2 & MD2 & !OD2 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (LD3 & !RD3 & !MD3 & OD3 & LVMASK & RVMASK & OVMASK & MVMASK)
    # (!LD3 & RD3 & MD3 & !OD3 & LVMASK & RVMASK & OVMASK & MVMASK);

LERR = (LEFT != CVOTED) & !QERR & !QERR0 & !QERR1;
RERR = (RIGHT != CVOTED) & !QERR & !QERR0 & !QERR1;
OERR = (OPP != CVOTED) & !QERR & !QERR0 & !QERR1;
MERR = (MINE != CVOTED) & !QERR & !QERR0 & !QERR1;

END VOTER
```

B.3.2 Debug Router

```
MODULE ROUTER
TITLE 'DATAP DEBUG ROUTER, 10 OCT 87'

ROUTE DEVICE 'P16L8';
MI0, MI1, LI0, LI1, RI0, RI1, OI0, OI1 PIN 1,2,3,4,5,6,7,8;
LO0, LO1, RO0, RO1, OO0, OO1 PIN  18,17,14,12,16,15;
LDBEN,RDBEN,ODBEN PIN 9,11,13;

MI = [MI1,MI0];
```

```
LI = [LI1,LI0];
RI = [RI1,RI0];
OI = [OI1,OI0];
LO = [LO1,LO0];
RO = [RO1,RO0];
OO = [OO1,OO0];
LD = [LDBEN,LDBEN];
RD = [RDBEN,RDBEN];
OD = [ODBEN,ODBEN];


EQUATIONS

LO = MI & LD
# LI & !LD;

RO = MI & RD
# RI & !RD;

OO = MI & OD
# OI & !OD;

END ROUTER
```

B.3.3 Synchronous Data Path Controller

```
MODULE SYNCDATACONT FLAG '-R3'
TITLE 'SYNCHRONOUS DATA PATH CONTROLLER, 9 OCT 87'

SYNC DEVICE 'P16R4';
CK8MHZ,NE8MHZ PIN 1,2;
MYBOUND,MYFTC PIN 3,4;
VOTEIN,F0,F1 PIN 5,6,7;
REFSEL0,REFSEL1,REFSELECT0,REFSELECT1 PIN 8,9,15,14;
NEID0,NEID1,REGF0,REGF1 PIN 13,12,17,16;
!S0,!SIMINE,GND PIN 19,18,11;

REFSEL = [REFSEL1,REFSEL0];
REFSELECT = [REFSELECT0,REFSELECT1];
NEID = [NEID1,NEID0];
A,B,C,D = 3,2,1,0;
```

```
EQUATIONS

REGF0 := F0 & !MYFTC
# REGF0 & MYFTC;

REGF1 := F1 & !MYFTC
# REGF1 & MYFTC;

SO = !(MYBOUND & NE8MHZ) & MYFTC & REGF1 & !REGF0 & !VOTEIN
  # VOTEIN & !NE8MHZ
  # F1 & F0 & !VOTEIN;

SIMINE = !(!REGF1 & !REGF0) & (MYBOUND & NE8MHZ) & MYBOUND;

REFSELECT := (NEID + REFSEL);

END SYNCDATACONT
```

B.3.4 Asynchronous Data Path Controller

```
MODULE ASYNCDATACONT
TITLE 'ASYNCHRONOUS DATA PATH CONTROLLER, 9 OCT 87'

ASYXX DEVICE 'E0900';
C8MHZ PIN 1;
SERP PIN 7;
MYDATACLK,MYBOUND,F0,F1,CLEARDATAP PIN 2,3,4,5,21;
RBOUND,LBOUND,OBOUND,RDBEN,LDBEN,ODBEN PIN 39,38,37,36,35,34;
RDS,LDS,ODS,RCS,LCS,OCS,RVLTN,LVLTN,OVLTN
                    PIN 33,32,31,30,29,24,17,18,19;
CLEARDATAPSEL0,CLEARDATAPSEL1 PIN 23,22;
CLEARMREQ,CLEARLREQ,CLEARRREQ,CLEAROREQ PIN 28,27,26,25;
!CLEARMINE,!CLEARRIGHT,!CLEARLEFT,!CLEAROPP PIN 6,14,15,16;
RREGSHIFTIN,LREGSHIFTIN,OREGSHIFTIN PIN 11,12,13;
!SIRIGHT,!SILEFT,!SIOPP PIN 8,9,10;

CLEAR = [CLEARDATAPSEL1,CLEARDATAPSEL0];
CLEARMREQ ISTYPE 'POS,REG,FEED_REG';
CLEARLREQ ISTYPE 'POS,REG,FEED_REG';
CLEARRREQ ISTYPE 'POS,REG,FEED_REG';
```

```
CLEAROREQ ISTYPE 'POS,REG,FEED_REG';
CLEARMINE,CLEARRIGHT,CLEARLEFT,CLEAROPP ISTYPE 'NEG,COM,FEED_PIN';
SIRIGHT,SILEFT,SIOPP ISTYPE 'NEG,COM,FEED_PIN';
LREGSHIFTIN,RREGSHIFTIN,OREGSHIFTIN ISTYPE 'POS,REG,FEED_REG';
LDBEN,RDBEN,ODBEN ISTYPE 'FEED_PIN';

EQUATIONS

SERP := (F0 & F1) & !MYBOUND
# SERP & MYBOUND;

LREGSHIFTIN := (F0 # F1) & (!LBOUND & !LDBEN)
# LREGSHIFTIN & (LBOUND & !LDBEN) & !CLEARLREQ & SERP
# LREGSHIFTIN & (LBOUND & !LDBEN) & !SERP
# (F0 # F1) & (!MYBOUND & LDBEN)
# LREGSHIFTIN & (MYBOUND & LDBEN);

RREGSHIFTIN := (F0 # F1) & (!RBOUND & !RDBEN)
# RREGSHIFTIN & (RBOUND & !RDBEN) & !CLEARRREQ & SERP
# RREGSHIFTIN & (RBOUND & !RDBEN) & !SERP
# (F0 # F1) & (!MYBOUND & RDBEN)
# RREGSHIFTIN & (MYBOUND & RDBEN);

OREGSHIFTIN := (F0 # F1) & (!OBOUND & !ODBEN)
# OREGSHIFTIN & (OBOUND & !ODBEN) & !CLEAROREQ & SERP
# OREGSHIFTIN & (OBOUND & !ODBEN) & !SERP
# (F0 # F1) & (!MYBOUND & ODBEN)
# OREGSHIFTIN & (MYBOUND & ODBEN);

SILEFT = LDS & LBOUND & LREGSHIFTIN & !LDBEN
# (LCS & LVLTN) & LBOUND & LREGSHIFTIN & !LDBEN
# MYDATACLK & MYBOUND & LREGSHIFTIN & LDBEN;

SIRIGHT = RDS & RBOUND & RREGSHIFTIN & !RDBEN
# (RCS & RVLTN) & RBOUND & RREGSHIFTIN & !RDBEN
# MYDATACLK & MYBOUND & RREGSHIFTIN & RDBEN;


SIOPP = ODS & OBOUND & OREGSHIFTIN & !ODBEN
# (OCS & OVLTN) & OBOUND & OREGSHIFTIN & !ODBEN
# MYDATACLK & MYBOUND & OREGSHIFTIN & ODBEN;
```

```
CLEARMINE = (CLEARMREQ & !MYBOUND);

CLEARMREQ := (CLEAR == 0) # (CLEARMREQ & !(CLEAR == 0));
CLEARMREQ.RE = CLEARMINE;

CLEARLEFT = (CLEARLREQ & !MYBOUND & LDBEN)
    # (CLEARLREQ & !LBOUND & !LDBEN)
    # (CLEARLREQ & !LREGSHIFTIN & !LDBEN);

CLEARLREQ := (CLEAR == 1) # (CLEARLREQ & !(CLEAR == 1));
CLEARLREQ.RE = CLEARLEFT;

CLEARRIGHT = (CLEARRREQ & !MYBOUND & RDBEN)
    # (CLEARRREQ & !RBOUND & !RDBEN)
    # (CLEARRREQ & !RREGSHIFTIN & !RDBEN);

CLEARRREQ := (CLEAR == 2) # (CLEARRREQ & !(CLEAR == 2));
CLEARRREQ.RE = CLEARRIGHT;

CLEAROPP = (CLEAROREQ & !MYBOUND & ODBEN)
    # (CLEAROREQ & !OBOUND & !ODBEN)
    # (CLEAROREQ & !OREGSHIFTIN & !ODBEN);

CLEAROREQ := (CLEAR == 3) # (CLEAROREQ & !(CLEAR == 3));
CLEAROREQ.RE = CLEAROPP;

END ASYNCDATACONT
```

B.3.5 Syndrome Accumulator

```
MODULE SYNACC
TITLE 'SYNDROME ACCUMULATOR PAL, 8 OCT 1987'

SYN DEVICE 'E0600';
NE8MHZ PIN 1;
LERR0,LERR1,RERR0,RERR1,OERR0,OERR1 PIN 2,11,23,14,15,16;
MERR0,MERR1,QERR PIN 17, 18, 20;
VOTEIN, VOTEOUT1, VOTEOUT PIN 8, 9, 10;
NERROR1READ PIN 22;
ACCLERR,ACCRERR,ACCOERR,ACCMERR,ACCQERR PIN 3, 4, 5, 7, 6;
VOTINGSERP PIN 21;
```

```
OUTPUTS = [ACCLERR,ACCRERR,ACCOERR,ACCMERR,ACCQERR];
LERR = [LERR1,LERR0];
RERR = [RERR1,RERR0];
OERR = [OERR1,OERR0];
MERR = [MERR1,MERR0];
CLK,VS,VI,VO1,VO = NE8MHZ,VOTINGSERP,VOTEIN,VOTEOUT1,VOTEOUT;

NER = NERROR1READ;
ERROR1READ = !NERROR1READ;
ER = ERROR1READ;
EQUATIONS

ACCLERR := (((LERR0 # LERR1) & VOTEOUT1 & !VOTINGSERP)
  # ACCLERR ) & NERROR1READ;

ACCRERR := (((RERR0 # RERR1) & VOTEOUT1 & !VOTINGSERP)
  # ACCRERR ) & NERROR1READ;

ACCOERR := (((OERR0 # OERR1) & VOTEOUT1 & !VOTINGSERP)
  # ACCOERR ) & NERROR1READ;

ACCMERR := (((MERR0 # MERR1) & VOTEOUT1 & !VOTINGSERP)
  # ACCMERR ) & NERROR1READ;

ACCQERR := ((QERR & VOTEOUT1 & !VOTINGSERP)
  # ACCQERR ) & NERROR1READ;


ENABLE OUTPUTS = ERROR1READ;
ENABLE VOTEOUT1 = 1;

VOTEOUT1 := VOTEIN;

VOTEOUT :=VOTEOUT1;

END SYNACC
```

B.3.6 Link Error Accumulator

```
MODULE LINKERRORACC
```

```
TITLE 'LINK ERROR ACCUMULATOR, 13 OCT 87'

LIN DEVICE 'E0600';
NE8MHZ PIN 1;
LEFTVLTN,RIGHTVLTN,OPPVLTN,!ERROR1READ PIN 2,11,23,14;
LLINKERROR,RLINKERROR,OLINKERROR PIN 3,4,5;
ACCLLINKERR,ACCRLINKERR,ACCOLINKERR PIN 6,7,8;

SET_TIMEOUT PIN 9;
SCORECNTCLK PIN 22;
SCALECNT0 PIN 15;
SCALECNT1 PIN 16;
SCALECNT2 PIN 17;
SCALECNT3 PIN 18;
SCALECNT4 PIN 19;
SCALECNT5 PIN 20;
SCALECNT6 PIN 21;
DECIDE PIN 13;

OUTPUTS = [ACCLLINKERR,ACCRLINKERR,ACCOLINKERR];

COUNT = [SCALECNT6,
SCALECNT5,
SCALECNT4,
SCALECNT3,
SCALECNT2,
SCALECNT1,
SCALECNT0];

VLTNS = [LEFTVLTN, RIGHTVLTN, OPPVLTN];

FUSES

"SET FUSE FOR LLINKERROR"
6441 = 1;
"SET FUSE FOR RLINKERROR"
6446 = 1;
"SET FUSE FOR OLINKERROR"
6451 = 1;

EQUATIONS
```

```
LLINKERROR := 1;
LLINKERROR.C = LEFTVLTN;
ACCLLINKERR := LLINKERROR;
LLINKERROR.RE = ERROR1READ;

RLINKERROR := 1;
RLINKERROR.C = RIGHTVLTN;
ACCRLINKERR := RLINKERROR;
RLINKERROR.RE = ERROR1READ;

OLINKERROR := 1;
OLINKERROR.C = OPPVLTN;
ACCOLINKERR := OLINKERROR;
OLINKERROR.RE = ERROR1READ;

ENABLE OUTPUTS = ERROR1READ;

COUNT := COUNT+1;
SCALECNT6.RE = SET_TIMEOUT;
SCALECNT5.RE = SET_TIMEOUT;
SCALECNT4.RE = SET_TIMEOUT;
SCALECNT3.RE = SET_TIMEOUT;
SCALECNT2.RE = SET_TIMEOUT;
SCALECNT1.RE = SET_TIMEOUT;
SCALECNT0.RE = SET_TIMEOUT;

SCORECNTCLK = (COUNT == ^B00000001);

END LINKERRORACC
```

B.4 Scoreboard

B.4.1 Scoreboard First Half

```
MODULE SCORE1
TITLE 'SCOREBOARD 1ST HALF, 25 OCT 87'

SCRE1 DEVICE 'E0900';

SCALECNT,DECIDE PIN 1,21;
```

```
PEAMASK,PEBMASK,PECMASK,PEDMASK PIN 39,2,3,4;
CLASS2,CLASS1,CLASS0,CTS PIN 17,18,19,22;
TIMEOUT,!READERROR2 PIN 29,23;
SCOREOP,NE8MHZ PIN 37,38;
DUPLEX,MAJ,DUPLEX_MAJ,UNANIMOUS,DUPLEX_UNANIMOUS PIN 5,6,7,8,9;
AOK,BOK,COK,DOK PIN 25,26,27,28;
SET_TIMEOUT,TIMEOUT_SET,GO PIN 36,35,34;
A_TIMEOUT_ERR,B_TIMEOUT_ERR PIN 33,32;
C_TIMEOUT_ERR,D_TIMEOUT_ERR PIN 31,30;
C0,C1,C2,C3,C4,C5,C6 PIN 10,11,12,13,14,15,16;
TO_ENABLED PIN 24;

TIMEOUT_ERRS = [A_TIMEOUT_ERR,B_TIMEOUT_ERR,
C_TIMEOUT_ERR,D_TIMEOUT_ERR];
COUNT = [C6,C5,C4,C3,C2,C1];


C0,C1,C2,C3,C4,C5,C6 ISTYPE 'REG,FEED_REG';
AOK,BOK,COK,DOK ISTYPE 'REG,FEED_REG';
MAJ,DUPLEX,DUPLEX_MAJ,UNANIMOUS,DUPLEX_UNANIMOUS
                            ISTYPE 'COM,FEED_PIN';
SET_TIMEOUT,TIMEOUT_SET,GO ISTYPE 'REG,FEED_REG';
TIMEOUT ISTYPE 'COM,FEED_PIN';

FUSES
17321 = 1;
17326 = 1;
17331 = 1;
17336 = 1;
"FUSE FOR TO_ENABLED

EQUATIONS

AOK := BOK;
AOK.C := SCOREOP & !NE8MHZ;

BOK := COK;
BOK.C := SCOREOP & !NE8MHZ;

COK := DOK;
COK.C := SCOREOP & !NE8MHZ;
```

```
DOK := CTS;
DOK.C := SCOREOP & !NE8MHZ;

MAJ = (AOK & BOK & PEAMASK & PEBMASK
  #AOK & COK & PEAMASK & PECMASK
#AOK & DOK & PEAMASK & PEDMASK
#BOK & COK & PEBMASK & PECMASK
#BOK & DOK & PEBMASK & PEDMASK
#COK & DOK & PECMASK & PEDMASK);

DUPLEX = PEAMASK & PEBMASK & !PECMASK & !PEDMASK
#PEAMASK & PECMASK & !PEBMASK & !PEDMASK
#PEAMASK & PEDMASK & !PEBMASK & !PECMASK
#PEBMASK & PECMASK & !PEAMASK & !PEDMASK
#PEBMASK & PEDMASK & !PEAMASK & !PECMASK
#PECMASK & PEDMASK & !PEAMASK & !PEBMASK;

DUPLEX_MAJ = (AOK & PEAMASK
# BOK & PEBMASK
# COK & PECMASK
# DOK & PEDMASK) & DUPLEX;

UNANIMOUS = AOK & BOK & COK & DOK & PEAMASK & PEBMASK & PECMASK & PEDMASK
   #AOK & BOK & COK & PEAMASK & PEBMASK & PECMASK & !PEDMASK
   #AOK & BOK & !PECMASK & DOK & PEAMASK & PEBMASK & PEDMASK
   #AOK & !PEBMASK & COK & DOK & PEAMASK & PECMASK & PEDMASK
   #!PEAMASK & BOK & COK & DOK & PEBMASK & PECMASK & PEDMASK;

DUPLEX_UNANIMOUS = AOK & BOK & !PECMASK & !PEDMASK & PEAMASK & PEBMASK
  #AOK & !PEBMASK & !PECMASK & DOK & PEAMASK & PEDMASK
  #!PEAMASK & !PEBMASK & COK & DOK & PECMASK & PEDMASK
  #!PEAMASK & BOK & COK & !PEDMASK & PEBMASK & PECMASK
  #AOK & !PEBMASK & COK & !PEDMASK & PEAMASK & PECMASK
  #!PEAMASK & BOK & !PECMASK & DOK & PEBMASK & PEDMASK;

SET_TIMEOUT := ((MAJ & !DUPLEX) # (DUPLEX_MAJ & DUPLEX))
 & (!(UNANIMOUS & !DUPLEX) # !(DUPLEX_UNANIMOUS & DUPLEX)) & !TIMEOUT_SET;

TIMEOUT_SET := SET_TIMEOUT # (TIMEOUT_SET & !TIMEOUT & !GO);

COUNT := (COUNT + 1) & TIMEOUT_SET
# 0 & !TIMEOUT_SET;
```

```
TIMEOUT = 0;
" TIMEOUT = (COUNT==1) & TIMEOUT_SET;

GO := (UNANIMOUS & !DUPLEX) # (DUPLEX & DUPLEX_UNANIMOUS)
 # (TIMEOUT & TIMEOUT_SET);

ENABLE GO = 1;

A_TIMEOUT_ERR := !AOK & TIMEOUT;

B_TIMEOUT_ERR := !BOK & TIMEOUT;

C_TIMEOUT_ERR := !COK & TIMEOUT;

D_TIMEOUT_ERR := !DOK & TIMEOUT;

ENABLE TIMEOUT_ERRS = READERROR2;

" TO_ENABLE := 1;
" TO_ENABLE.C = GO;
" TO_ENABLE.RE = ;

END SCORE1
```

B.4.2 Scoreboard Second Half

```
MODULE SCORE2
TITLE 'SCOREBOARD 2ND HALF, 27 OCT 87'

SCRE2 DEVICE 'E0900';
SCOREOP,NE8MHZ PIN 37,38;
PEAMASK,PEBMASK,PECMASK,PEDMASK PIN 39,2,3,4;
I0,I1,I2,I3,I4,I5,I6 PIN 17,18,19,22,23,24,25;
A0,A1,A2,A3,A4,A5,A6 PIN 5,6,7,8,9,10,11;
B0,B1,B2,B3,B4,B5,B6 PIN 12,13,14,15,16,26,27;
C0,C1,C2,C3,C4,C5,C6 PIN 28,29,30,31,32,33,34;
S0,S1 PIN 35,36;

S = [S1,S0];
A = [A6,A5,A4,A3,A2,A1,A0];
```

```
B = [B6,B5,B4,B3,B2,B1,B0];
C = [C6,C5,C4,C3,C2,C1,C0];
I = [I6,I5,I4,I3,I2,I1,I0];
AMASK = [PEAMASK,PEAMASK,PEAMASK,PEAMASK,PEAMASK,PEAMASK,PEAMASK];
BMASK = [PEBMASK,PEBMASK,PEBMASK,PEBMASK,PEBMASK,PEBMASK,PEBMASK];
CMASK = [PECMASK,PECMASK,PECMASK,PECMASK,PECMASK,PECMASK,PECMASK];
DMASK = [PEDMASK,PEDMASK,PEDMASK,PEDMASK,PEDMASK,PEDMASK,PEDMASK];

FUSES
17341 = 1;
17346 = 1;
17351 = 1;
17356 = 1;
17361 = 1;
17366 = 1;
17371 = 1;
17376 = 1;
17381 = 1;
17386 = 1;
17391 = 1;
17396 = 1;
17331 = 1;
17326 = 1;
17321 = 1;
17316 = 1;
17311 = 1;
17306 = 1;
17301 = 1;
17296 = 1;
17291 = 1;
17286 = 1;
17281 = 1;

EQUATIONS

S := (S + 1);
S1.C = SCOREOP & !NE8MHZ;
S0.C = SCOREOP & !NE8MHZ;

C := I;
C6.C = SCOREOP & !NE8MHZ;
C5.C = SCOREOP & !NE8MHZ;
```

```
C4.C = SCOREOP & !NE8MHZ;
C3.C = SCOREOP & !NE8MHZ;
C2.C = SCOREOP & !NE8MHZ;
C1.C = SCOREOP & !NE8MHZ;
C0.C = SCOREOP & !NE8MHZ;

B := C,
B6.C = SCOREOP & !NE8MHZ;
B5.C = SCOREOP & !NE8MHZ;
B4.C = SCOREOP & !NE8MHZ;
B3.C = SCOREOP & !NE8MHZ;
B2.C = SCOREOP & !NE8MHZ;
B1.C = SCOREOP & !NE8MHZ;
B0.C = SCOREOP & !NE8MHZ;

A := (B & !(S == 3))
   #(((A & AMASK & B & BMASK)
     #(A & AMASK & C & CMASK)
     #(A & AMASK & I & DMASK)
     #(B & BMASK & C & CMASK)
     #(B & BMASK & I & DMASK)
     #(C & CMASK & I & DMASK))
    & (S == 3));
A6.C = SCOREOP & !NE8MHZ;
A5.C = SCOREOP & !NE8MHZ;
A4.C = SCOREOP & !NE8MHZ;
A3.C = SCOREOP & !NE8MHZ;
A2.C = SCOREOP & !NE8MHZ;
A1.C = SCOREOP & !NE8MHZ;
A0.C = SCOREOP & !NE8MHZ;
END SCORE2
```

B.5 Global Controller

B.5.1 Multiplexor First Half

```
MODULE CONTMUX1
TITLE 'GLOBAL CONTROLLER CONDITIONAL MUX UPPER HALF, 19 NOV 87'

MUX1 DEVICE 'P22V10';
```

```
NE16MHZ PIN 1;
CDONE,MYFTC,NE8MHZDEL PIN 2,3,4;
GO,VDO,VD1,VD2,VD3,VD4 PIN 5,6,7,13,20,21;
SELO,SEL1,SEL2,SEL3 PIN 8,9,10,11;
LSELO,LSEL1,LSEL2,LSEL3 PIN 23,22,15,14;
SELMODE,MYINPUT PIN 16,17;
CONDITION PIN 18;

SEL = [SELO,SEL1,SEL2,SEL3];
LSEL = [LSELO,LSEL1,LSEL2,LSEL3];


EQUATIONS

CONDITION := CDONE & (SEL == 0) & SELMODE
# CDONE & (LSEL == 0) & !SELMODE
       # MYFTC & (SEL == 1) & SELMODE
# MYFTC & (LSEL == 1) & !SELMODE
    # NE8MHZDEL & (SEL == 2) & SELMODE
# NE8MHZDEL & (LSEL == 2) & !SELMODE
  # GO & (SEL == 3) & SELMODE
# GO & (LSEL == 3) & !SELMODE
# VDO & (LSEL == 4) & !SELMODE
# VD1 & (LSEL == 5) & !SELMODE
# VD2 & (LSEL == 6) & !SELMODE
# VD3 & (LSEL == 7) & !SELMODE
# VD4 & (LSEL == 8) & !SELMODE
# MYINPUT & ((LSEL == 14) # (LSEL == 15)) & !SELMODE;

LSEL := SEL & SELMODE
# LSEL & !SELMODE;

END CONTMUX1
```

B.5.2 Multiplexor Second Half

```
MODULE CONTMUX2
TITLE 'GLOBAL CONTROLLER CODITIONAL MUX LOWER HALF, 19 NOV 87'

MUX2 DEVICE 'P22V10';
```

```
NE16MHZ PIN 1;
VD0,VD1,VD2,VD3,VD4 PIN 2,3,4,5,6;
EXCLASS0,EXCLASS1,EXCLASS2 PIN 7,8,9;
LADD,DLADD,D2LADD,AUTOINCREMENT PIN 10,17,18,11;
LSEL0,LSEL1,LSEL2,LSEL3 PIN 16,15,14,13;
MYINPUT,CLADD PIN 23,22;
ICNT2,ICNT1,ICNT0 PIN 19,20,21;

COUNT = [ICNT2,ICNT1,ICNT0];
LATCHEDSEL = [LSEL0,LSEL1,LSEL2,LSEL3];

EQUATIONS

MYINPUT = VD4 & (COUNT == 0) & (LATCHEDSEL == 14)
# VD3 & (COUNT == 1) & (LATCHEDSEL == 14)
# VD2 & (COUNT == 2) & (LATCHEDSEL == 14)
# VD1 & (COUNT == 3) & (LATCHEDSEL == 14)
# VD0 & (COUNT == 4) & (LATCHEDSEL == 14)
# EXCLASS2 & (COUNT == 0) & (LATCHEDSEL == 15)
# EXCLASS1 & (COUNT == 1) & (LATCHEDSEL == 15)
# EXCLASS0 & (COUNT == 2) & (LATCHEDSEL == 15);

COUNT := (COUNT + 1) & (LATCHEDSEL == 14) & (COUNT != 4)
#(COUNT + 1) & (LATCHEDSEL == 15) & (COUNT != 2);

DLADD := LADD;

D2LADD := DLADD;

CLADD = LADD & !DLADD & !NE16MHZ
# LADD & D2LADD & NE16MHZ;

END CONTMUX2
```

B.5.3 Next State Register

```
MODULE NSREG
TITLE 'GLOBAL CONTROLLER NEXT STATE REGISTER, 19 NOV 87'

NSTATEREG DEVICE 'P22V10';
```

```
CLADD,NSSELECT PIN 1,13;
CT0,CT1,CT2,CT3,CT4 PIN 2,3,4,5,6;
CT5,CT6,CT7,CT8,CT9 PIN 7,8,9,10,11;
NS0,NS1,NS2,NS3,NS4 PIN 23,22,21,20,15;
NS5,NS6,NS7,NS8,NS9 PIN 16,17,18,19,14;

CT = [CT8,CT7,CT6,CT5,CT4,CT3,CT2,CT1,CT0];
NS = [NS8,NS7,NS6,NS5,NS4,NS3,NS2,NS1,NS0];
SNS = [NS7,NS6,NS5,NS4,NS3,NS2,NS1,NS0];


EQUATIONS

NS := (CT + 1) & !NSSELECT;

SNS := (SNS + 1) & NSSELECT;

NS8 := NS7 & NS6 & NS5 & NS4 & NS3 & NS2 & NS1 & NS0
& !NS8 & NSSELECT
# NS8 & NSSELECT;

ENABLE NS = NSSELECT;

NS9 := CT8 & CT7 & CT6 & CT5 & CT4 & CT3 & CT2 & CT1 & CT0 & !CT9 & NSSELECT
# CT9 & !NSSELECT
# NS9 & NSSELECT;

ENABLE NS9 = NSSELECT;

END NSREG
```

B.5.4 Controller Decoder

```
MODULE DECODE
TITLE 'GLOBAL CONTROLLER OUTPUT DECODER, 5 JAN 88';

CDECODE DEVICE 'P22V10';

NE16MHZBAR PIN 1;
SEL0,SEL1,SEL2,SEL3,SELMODE PIN 2,3,4,5,6;
```

```
VMASKLATCH,EXADDRLATCH,MDLATCH,CTLVDATAC PIN 14,15,16,17;
!KICKDOG,DECIDE,!RESETREC,!RESETXMIT PIN 18,19,20,21;
CLEARDATAP,CTLCTRCLK PIN 22,23;

SEL = [SEL0,SEL1,SEL2,SEL3];

EQUATIONS

EXADDRLATCH := !SELMODE & (SEL == 3);

MDLATCH := !SELMODE & (SEL == 4);

CTLVDATAC := !SELMODE & (SEL == 2);

KICKDOG := !SELMODE & (SEL == 1);

DECIDE := !SELMODE & (SEL == 6);

RESETREC := !SELMODE & (SEL == 7);

RESETXMIT := !SELMODE & (SEL == 8);

CLEARDATAP := !SELMODE & (SEL == 9);

CTLCTRCLK := !SELMODE & (SEL == 5);

VMASKLATCH := !SELMODE & (SEL == 10);

END DECODE
```

B.5.5 Mask, Size, and Debug Register

```
MODULE MASKREG FLAG '-R3'
TITLE 'GLOBAL CONTROLLER MASK REGISTER/XLATER, 11 NOV 87'

MREG DEVICE 'E0900';

NEID0,NEID1,REGLATCH,REGLATCHA,LF0 PIN 2,3,1,21,4;
VD0,VD1,VD2,VD3,VD4,VD5,VD6,VD7 PIN 17,18,19,22,23,24,37,38;
CMASKL,CMASKR,CMASK0 PIN 25,26,27;
NEMASKA,NEMASKB,NEMASKC,NEMASKD PIN 33,34,35,36;
```

```
PEMASKA,PEMASKB,PEMASKC,PEMASKD PIN 5,6,7,8;
LDBEN,RDBEN,ODBEN,VDBEN PIN 9,10,11,12;
PSIZE3,PSIZE2,PSIZE1,PSIZE0 PIN 13,14,15,16;
SIZE3,SIZE2,SIZE1,SIZE0 PIN 30,31,32,39;
CLRSEL0,CLRSEL1 PIN 29,28;

DEBUGINFO = [LDBEN,RDBEN,ODBEN,VDBEN];
VDATAU = [VD7,VD6,VD5,VD4];
 VDATAL = [VD3,VD2,VD1,VD0];
LF = [LF0,VD7,VD6,VD5];
PSIZE = [PSIZE3,PSIZE2,PSIZE1,PSIZE0];
SIZE = [SIZE3,SIZE2,SIZE1,SIZE0];
PEMASK = [PEMASKA,PEMASKB,PEMASKC,PEMASKD];
NEMASK = [NEMASKA,NEMASKB,NEMASKC,NEMASKD];
NEID = [NEID1,NEID0];
CLEARREQ = [VD1,VD0];


EQUATIONS

DEBUGINFO := VDATAL & (LF == 12)
  # DEBUGINFO & !(LF == 12);

CMASKL := VD3 & (LF == 9)
    # VD3 & (NEID == 1) & (!LF0)
# VD2 & (NEID == 2) & (!LF0)
# VD1 & (NEID == 3) & (!LF0)
# VD0 & (NEID == 0) & (!LF0)
# CMASKL & !(LF == 9) & LF0;

CMASKR := VD2 & (LF == 9)
    # VD3 & (NEID == 3) & (!LF0)
# VD2 & (NEID == 0) & (!LF0)
# VD1 & (NEID == 1) & (!LF0)
# VD0 & (NEID == 2) & (!LF0)
# CMASKR & !(LF == 9) & LF0;

CMASKO := VD1 & (LF == 9)
    # VD3 & (NEID == 2) & (!LF0)
# VD2 & (NEID == 3) & (!LF0)
# VD1 & (NEID == 0) & (!LF0)
# VD0 & (NEID == 1) & (!LF0)
```

```
# CMASKO & !(LF == 9) & LFO;


PSIZE := VDATAL & (LF == 11)
    # SIZE & (LF == 10)
    # PSIZE & !(LF == 11) & !(LF == 10);

PEMASK := VDATAU & VDATAL & !LFO
# PEMASK & LFO;

NEMASK := VDATAL & !LFO
# NEMASK & LFO;

CLRSEL0 := 0 & (NEID == 0) & (CLEARREQ == 0) & (LF == 13)
# 0 & (NEID == 0) & (CLEARREQ == 1) & (LF == 13)
# 1 & (NEID == 0) & (CLEARREQ == 2) & (LF == 13)
# 1 & (NEID == 0) & (CLEARREQ == 3) & (LF == 13)
# 1 & (NEID == 1) & (CLEARREQ == 0) & (LF == 13)
# 0 & (NEID == 1) & (CLEARREQ == 1) & (LF == 13)
# 0 & (NEID == 1) & (CLEARREQ == 2) & (LF == 13)
# 1 & (NEID == 1) & (CLEARREQ == 3) & (LF == 13)
# 1 & (NEID == 2) & (CLEARREQ == 0) & (LF == 13)
# 1 & (NEID == 2) & (CLEARREQ == 1) & (LF == 13)
# 0 & (NEID == 2) & (CLEARREQ == 2) & (LF == 13)
# 0 & (NEID == 2) & (CLEARREQ == 3) & (LF == 13)
# 0 & (NEID == 3) & (CLEARREQ == 0) & (LF == 13)
# 1 & (NEID == 3) & (CLEARREQ == 1) & (LF == 13)
# 1 & (NEID == 3) & (CLEARREQ == 2) & (LF == 13)
# 0 & (NEID == 3) & (CLEARREQ == 3) & (LF == 13);

CLRSEL1 := 0 & (NEID == 0) & (CLEARREQ == 0) & (LF == 13)
# 1 & (NEID == 0) & (CLEARREQ == 1) & (LF == 13)
# 1 & (NEID == 0) & (CLEARREQ == 2) & (LF == 13)
# 0 & (NEID == 0) & (CLEARREQ == 3) & (LF == 13)
# 0 & (NEID == 1) & (CLEARREQ == 0) & (LF == 13)
# 0 & (NEID == 1) & (CLEARREQ == 1) & (LF == 13)
# 1 & (NEID == 1) & (CLEARREQ == 2) & (LF == 13)
# 1 & (NEID == 1) & (CLEARREQ == 3) & (LF == 13)
# 1 & (NEID == 2) & (CLEARREQ == 0) & (LF == 13)
# 0 & (NEID == 2) & (CLEARREQ == 1) & (LF == 13)
# 0 & (NEID == 2) & (CLEARREQ == 2) & (LF == 13)
# 1 & (NEID == 2) & (CLEARREQ == 3) & (LF == 13)
```

```
# 1 & (NEID == 3) & (CLEARREQ == 0) & (LF == 13)
# 1 & (NEID == 3) & (CLEARREQ == 1) & (LF == 13)
# 0 & (NEID == 3) & (CLEARREQ == 2) & (LF == 13)
# 0 & (NEID == 3) & (CLEARREQ == 3) & (LF == 13);

END MASKREG
```

B.5.6 Data Path Voter Mask Register

```
MODULE VOTEMASK FLAG '-R3'
TITLE 'GLOBAL CONTROLLER VOTER MASK REGISTER, 12 NOV 87';

VMASK DEVICE 'P22V10';

VMASKLATCH PIN 1;
NEID0,NEID1 PIN 2,3;
PEMASKA,PEMASKB,PEMASKC,PEMASKD PIN 4,5,6,7;
NEMASKA,NEMASKB,NEMASKC,NEMASKD PIN 8,9,10,11;
CTLBUS2,CTLBUS1,CTLBUS0 PIN 13,14,15;
VMASKL,VMASKR,VMASKO,VMASKM PIN 17,18,19,20;
!CLASSFIFOOE,CLASSFIFOOR,!RECE PIN 23,22,21;
SD07 PIN 16;

NEID = [NEID1,NEID0];
LF = [CTLBUS2,CTLBUS1,CTLBUS0];

EQUATIONS

VMASKL := NEMASKA & (NEID == 1) & (LF == 4) "1ROUND AM B
# NEMASKB & (NEID == 2) & (LF == 4) "1ROUND AM C
# NEMASKC & (NEID == 3) & (LF == 4) "1ROUND AM D
# NEMASKD & (NEID == 0) & (LF == 4) "1ROUND AM A
# 0 & (NEID == 1) & (LF == 0) "FROM A AM B
# (NEMASKB) & (NEID == 2) & (LF == 0) "FROMA AM C
# (NEMASKC) & (NEID == 3) & (LF == 0) "FROMA AM D
# (NEMASKD) & (NEID == 0) & (LF == 0) "FROMA AM A
# (NEMASKA) & (NEID == 1) & (LF == 1) "FROMB AM B
# 0 & (NEID == 2) & (LF == 1) "FROMB AM C
# (NEMASKC) & (NEID == 3) & (LF == 1) "FROMB AM D
# (NEMASKD) & (NEID == 0) & (LF == 1) "FROMB AM A
# (NEMASKA) & (NEID == 1) & (LF == 2) "FROMC AM B
```

```
# (NEMASKB) & (NEID == 2) & (LF == 2) "FROMC AM C
# 0 & (NEID == 3) & (LF == 2) "FROMC AM C
# (NEMASKD) & (NEID == 0) & (LF == 2) "FROMC AM A
# (NEMASKA) & (NEID == 1) & (LF == 3) "FROMD AM B
# (NEMASKB) & (NEID == 2) & (LF == 3) "FROMD AM C
# (NEMASKC) & (NEID == 3) & (LF == 3) "FROMD AM D
# 0 & (NEID == 0) & (LF == 3); "FROMD AM A

VMASKR := NEMASKA & (NEID == 3) & (LF == 4) "1ROUND AM D
# NEMASKB & (NEID == 0) & (LF == 4) "1ROUND AM A
# NEMASKC & (NEID == 1) & (LF == 4) "1ROUND AM B
# NEMASKD & (NEID == 2) & (LF == 4) "1ROUND AM C
# 0 & (NEID == 3) & (LF == 0) "FROM A AM D
# (NEMASKB) & (NEID == 0) & (LF == 0) "FROMA AM A
# (NEMASKC) & (NEID == 1) & (LF == 0) "FROMA AM B
# (NEMASKD) & (NEID == 2) & (LF == 0) "FROMA AM C
# (NEMASKA) & (NEID == 3) & (LF == 1) "FROMB AM D
# 0 & (NEID == 0) & (LF == 1) "FROMB AM A
# (NEMASKC) & (NEID == 1) & (LF == 1) "FROMB AM B
# (NEMASKD) & (NEID == 2) & (LF == 1) "FROMB AM C
# (NEMASKA) & (NEID == 3) & (LF == 2) "FROMC AM D
# (NEMASKB) & (NEID == 0) & (LF == 2) "FROMC AM A
# 0 & (NEID == 1) & (LF == 2) "FROMC AM B
# (NEMASKD) & (NEID == 2) & (LF == 2) "FROMC AM C
# (NEMASKA) & (NEID == 3) & (LF == 3) "FROMD AM D
# (NEMASKB) & (NEID == 0) & (LF == 3) "FROMD AM A
# (NEMASKC) & (NEID == 1) & (LF == 3) "FROMD AM B
# 0 & (NEID == 2) & (LF == 3); "FROMD AM C


VMASKO := NEMASKA & (NEID == 2) & (LF == 4) "1ROUND AM C
# NEMASKB & (NEID == 3) & (LF == 4) "1ROUND AM D
# NEMASKC & (NEID == 0) & (LF == 4) "1ROUND AM A
# NEMASKD & (NEID == 1) & (LF == 4) "1ROUND AM B
# 0 & (NEID == 2) & (LF == 0) "FROM A AM C
# (NEMASKB) & (NEID == 3) & (LF == 0) "FROMA AM D
# (NEMASKC) & (NEID == 0) & (LF == 0) "FROMA AM A
# (NEMASKD) & (NEID == 1) & (LF == 0) "FROMA AM B
# (NEMASKA) & (NEID == 2) & (LF == 1) "FROMB AM C
# 0 & (NEID == 3) & (LF == 1) "FROMB AM D
# (NEMASKC) & (NEID == 0) & (LF == 1) "FROMB AM A
# (NEMASKD) & (NEID == 1) & (LF == 1) "FROMB AM B
```

```
# (NEMASKA) & (NEID == 2) & (LF == 2) "FROMC AM C
# (NEMASKB) & (NEID == 3) & (LF == 2) "FROMC AM D
# 0 & (NEID == 0) & (LF == 2) "FROMC AM A
# (NEMASKD) & (NEID == 1) & (LF == 2) "FROMC AM B
# (NEMASKA) & (NEID == 2) & (LF == 3) "FROMD AM C
# (NEMASKB) & (NEID == 3) & (LF == 3) "FROMD AM D
# (NEMASKC) & (NEID == 0) & (LF == 3) "FROMD AM A
# 0 & (NEID == 1) & (LF == 3); "FROMD AM B


VMASKM := NEMASKA & (NEID == 0) & (LF == 4) "1ROUND AM A
# NEMASKB & (NEID == 1) & (LF == 4) "1ROUND AM B
# NEMASKC & (NEID == 2) & (LF == 4) "1ROUND AM C
# NEMASKD & (NEID == 3) & (LF == 4) "1ROUND AM D
# 0 & (NEID == 0) & (LF == 0) "FROM A AM A
# (NEMASKB) & (NEID == 1) & (LF == 0) "FROMA AM B
# (NEMASKC) & (NEID == 2) & (LF == 0) "FROMA AM C
# (NEMASKD) & (NEID == 3) & (LF == 0) "FROMA AM D
# (NEMASKA) & (NEID == 0) & (LF == 1) "FROMB AM A
# 0 & (NEID == 1) & (LF == 1) "FROMB AM B
# (NEMASKC) & (NEID == 2) & (LF == 1) "FROMB AM C
# (NEMASKD) & (NEID == 3) & (LF == 1) "FROMB AM D
# (NEMASKA) & (NEID == 0) & (LF == 2) "FROMC AM A
# (NEMASKB) & (NEID == 1) & (LF == 2) "FROMC AM B
# 0 & (NEID == 2) & (LF == 2) "FROMC AM C
# (NEMASKD) & (NEID == 3) & (LF == 2) "FROMC AM D
# (NEMASKA) & (NEID == 0) & (LF == 3) "FROMD AM A
# (NEMASKB) & (NEID == 1) & (LF == 3) "FROMD AM B
# (NEMASKC) & (NEID == 2) & (LF == 3) "FROMD AM C
# 0 & (NEID == 3) & (LF == 3); "FROMD AM D

SD07 = !RECE & CLASSFIFOOR;
ENABLE SD07 = CLASSFIFOOE;

END VOTEMASK
```

B.5.7 Global Controller Event Counter

```
MODULE GLOBCNT FLAG '-R3'
TITLE 'GLOBAL CONTROLLER COUNTER, 11 NOV 87'
```

```
COUNTER DEVICE 'P22V10';

CNTCLK PIN 1;
CTLBUS0,CTLBUS1,CTLBUS2,CTLBUS3 PIN 2,3,4,5;
SIZE0,SIZE1,SIZE2,SIZE3 PIN 6,7,8,9;
LOAD0,LOAD1 PIN 10,11;
C7,C6,C5,C4,C3,C2,C1,C0 PIN 19,22,21,20,23,18,17,16;
COUNTDONE PIN 15;

COUNTU = [C7,C6,C5,C4];
COUNTL = [C3,C2,C1,C0];
LOAD = [LOAD1,LOAD0];
CTL = [CTLBUS3,CTLBUS2,CTLBUS1,CTLBUS0];
SIZE = [SIZE3,SIZE2,SIZE1,SIZE0];
ZERO = [0,0,0,0];

EQUATIONS

COUNTL := (COUNTL - 1) & (LOAD == 0)
# ZERO & (LOAD == 1)
# CTL & (LOAD == 2)
# COUNTL & (LOAD == 3);

COUNTU := COUNTU & !(COUNTL == 0) & (LOAD == 0)
# (COUNTU - 1) & (COUNTL == 0) & (LOAD == 0)
# SIZE & (LOAD == 1)
# COUNTU & (LOAD == 2)
# CTL & (LOAD == 3);

COUNTDONE = (COUNTU == 0) & (COUNTL == 0);

END GLOBCNT
```

Bibliography

[AK84] A. Avizienis and J. Kelly. Fault tolerance by design diversity: concepts and experiments. *IEEE Computer*, August 1984.

[Bro87] L. Brock. *Oblique Wing Flight Control System.* Technical Report, C. S. Draper Laboratory, August 1987.

[DDS84] D. Dolev, C. Dwork, and L. Stockmeyer. *On the Minimal Synchronism Needed for Distributed Consensus.* IBM Research Report RJ 4292(46990), IBM, May 1984.

[Dol82] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3:14–30, 1982.

[FL82] M. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Fri86] S. A. Friend. *Process Synchronization Within a Loosely Coupled Fault Tolerant Parallel Processing System.* Master's thesis, Northeastern University, December 1986.

[GADS88] R. J. Gauthier, L. S. Alger, M. J. Dzwonczyk, and J. T. Sims. *System Architecture Evaluation and Design and Mission Management Functional Development.* Technical Report, C. S. Draper Laboratory, March 1988.

[Gol84] J. Goldberg. *Development and Analysis of the SIFT Computer.* NASA Contract Report 172146, SRI International, February 1984.

[Gra79] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, pages 393–481, Springer-Verlag, 1979.

[Har87] R. E. Harper. *Critical Issues in Ultra-Reliable Parallel Processing.* PhD thesis, Massachusetts Institute of Technology, June 1987.

[HSL78] A. L. Hopkins, T. B. Smith, and J. H. Lala. FTMP - a highly reli able fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10), October 1978.

[Joh88] S. C. Johnson. A methodology for reliability analysis of fault-tolerant parallel processor architectures over long space missions. to be published, February 1988.

[Kim75] C. R. Kime. Fault tolerant computing: an introduction and perspective. *IEEE Trans. Computers*, C-24(5):457–460, May 1975.

[KLJ85] J. Knight, N. Leveson, and L. St. Jean. A large scale experiment in N-version programming. In *Digest of Papers FTCS-15*, The 15th Annual International Symposium on Fault Tolerant Computing, Ann Arbor Michigan, June 1985.

[KSB85] C. M. Krishna, K. G. Shin, and R. W. Butler. Ensuring fault tolerance of phase locked clocks. *IEEE Trans. Computers*, C-34(8):752–756, August 1985.

[KWFT88] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Trans. Computers*, 37(4):398–405, April 1988.

[LA88] J. H. Lala and L. S. Alger. Hardware and software fault tolerance: a unified architectural approach. In *Digest of Papers FTCS-18*, The 18th Annual International Symposium on Fault Tolerant Computing, Tokoyo Japan, June 1988.

[LAGD86] J. H. Lala, L. S. Alger, R. J. Gauthier, and M. J. Dzwonczyk. A fault tolerant processor to meet rigorous failure requirements. In *Digest of Papers*, IEEE/AIAA 7th Digital Avionics System Conference, Fort Worth Texas, October 1986.

[Lal84] J. H. Lala. *Advanced Information Processing System (AIPS) System Specification.* Technical Report, C. S. Draper Laboratory, May 1984.

[Lal86] J. H. Lala. A byzantine resilient fault tolerant computer for nuclear power plant applications. In *Digest of Papers*, 16th Annual International Symosium on Fault Tolerant Computing Systems, Vienna Austria, July 1986.

[LL82] M. Pease L. Lamport, S. Shostak. The Byzantine generals problem. *ACM Trans. Programmig Languages and Systems*, 4(3):382–401, July 1982.

[LS88] Y-H. Lee and K. G. Shin. Optimal design and use of retry in fault-tolerant computer systems. *Journal of the ACM*, 35(1):45–69, January 1988.

[MG78] D. L. Martin and D. Gangsas. Testing of the YC-14 flight control system software. *AIAA Journal of Guidance and Control*, 1(4), July–August 1978.

[PB86] D. L. Palumbo and R. W. Butler. A performance evaluation of the software impemented fault-tolerance computer. *Jet Guidance*, 9(2):175–180, March–April 1986.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[SL84] K. G. Shin and Y-H. Lee. Error detection process-mode, design, and its impact on performance. *IEEE Trans. Computers*, C-33(6):529–540, June 1984.

[Smi83] T. B. Smith. *Fault Tolerant Processor Concepts and Operation*. Technical Report CSDL-P-1727, C. S. Draper Laboratory, May 1983.

[SS82] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford MA, 1982.

[Sta85] W. Stallings. *Data and Computer Communications*. Macmillan Publishing, New York NY, 1985.

[Wen78] J. Wensey. SIFT: the design and analysis of a fault-tolerant computer for aircraft control. *Proc IEEE*, 66:1240–1255, October 1978.

[WKF85] C. J. Walter, R. M. Kieckhafer, and A. M. Finn. MAFT: a multicomputer architecture for fault-tolerance in real-time systems. *Proc. IEEE Real Time Systems Symposium*, December 1985.