intellitech

The Intelligent Use of Technology

(2)

SIMULATION OF A FAULT TOLERANT MULTIPROCESSOR SYSTEM

Volume 1

INT-85-10

SIMULATION OF A FAULT TOLERANT MULTIPROCESSOR SYSTEM

Volume 1

MARCH 1985

Prepared By: Max Streit

Approved By: Dr. S.A. Mahmoud

INTELLITECH CANADA LIMITED

352 MacLaren Street
Ottawa, Ontario
K2P OM6

Government of Canada | Gouvernement du Canada

Department of Communications

DOC CONTRACTOR REPORT DOC-CR-SP 85-004

DEPARTMENT OF COMMUNICATIONS - OTTAWA - CANADA

SPACE PROGRAM

TITLE: Simulation of a Fault Tolerant Multiprocessor System

AUTHOR(S): Max Streit
INTELLITECH CANADA LIMITED
352 MacLaren Street
Ottawa, Ontario

ISSUED BY CONTRACTOR AS REPORT NO: INT-85-10

PREPARED BY: Max Streit

DEPARTMENT OF SUPPLY AND SERVICES CONTRACT NO: OER 83-05075

DOC SCIENTIFIC AUTHORITY: Michel Savoie
Communications Research Centre
Ottawa, Ontario

CLASSIFICATION: Unclassified

DATE: MARCH 1985

SUMMARY

The simulation activity described in this report is part of an overall study which includes the design and simulation of a fault tolerant multiprocessor architecture for spacecraft applications. The main focus of the overall study, however, has been on the development and utilization of computer aided engineering (CAE) design tools. One of these CAE design tools, known as N.mPc, is currently running on the VAX/VMS 11/780 system located in the Analysis and Simulation Laboratory of the Communications Research Centre, Department of Communications. The N.mPc package is used as a simulation environment for the work reported here.

N.mPc consists of a number of components used to describe the hardware behaviour of a target system, and to execute the simulation of that system.

The approach chosen for the design of the fault tolerant multiprocessor system is novel in the sense that fault tolerant features and supporting mechanisms are embedded in both the hardware architecture and the operating system software. The hardware has multiple redundant components that are controlled by fault detection mechanisms in order to prevent the propagation of errors from the faulty component to the remainder of the system. The operating system software contains all the intelligence needed to detect errors, identify their sources, take the necessary action to remove faulty units, reallocate the processing tasks and reconfigure the system to adapt to the new operational state.

Examples of the hardware faults which can be tolerated (ie. recovered from) by the full fault tolerant multiprocessor system include:

- Malfunction of a central processor.
- Malfunction of an interface processor.
- Failure of one of the redundant buses between interface and peripheral processors.
- A failing peripheral processor (device) is simply cut off from the redundant bus by its guardians.
- Failure of a gate complex (guardians, gates).
- Errors in received data due to hardware failures.

The CAE tool N.mPc is used for functional testing of the fault tolerant multiprocessor architecture by first developing descriptions of all necessary hardware modules in N.mPc´s hardware description language ISP´. The next step is to describe an interconnection scheme for the simulated hardware components of the multiprocessor architecture. Then some test software for the programmable hardware modules (descriptions of the Intel 8086 microprocessor) has to be developed. Finally simulated hardware and test software for the microprocessors are integrated under N.mPc to form an executable simulation that allows functional testing of the fault tolerant multiprocessor architecture and of the behaviour of the multiprocessor system in response to certain failures.

The originally planned integration of the simulated multiprocessor architecture and the full fault tolerant operating system was subsequently scoped down to a simulation involving the integration of

the fully simulated multiprocessor hardware architecture and only a subset of the operating system routines for the following reasons:

- The slow execution speed of N.mPc mitigates against conducting fault tolerant tests in a reasonable time period. This is due to the fact that the execution of each single operating system instruction involves detailed and numerous register transfer level instruction executions with N.mPc. Since the task execution cycles of the operating system are fairly lengthy to begin with, the corresponding number of executions within N.mPc will be extremely large, which leads to prohibitively long simulation times.
- The full operating system code, written in C programming language, proved to be lengthy and complex which makes the debugging process of the code, when executed under N.mPc, intractable.
- The N.mPc code corresponding to the VLSI structure of the 8086 chip details is not guaranteed to be absolutely correct. This complicates the debugging process and makes it difficult to determine whether the source of a given bug is in the VLSI description or in the complex operating system code. Note that this problem is inherent to any complex VLSI design such as in the 8086 processor.

In summary a full integration of the fault tolerant hardware and the operating system software proved to be beyond the time and resources allocated for this study.

A number of conclusions could be drawn from the major findings in this work:

1) When comparing the performance of the fault tolerant multiprocessor system described in this report to the performance of a system without fault tolerant features, we observe that the fault tolerant features add considerable redundancy and overhead due to continuous message exchanges and voting. Hence system throughput and processing power are reduced proportionately. This reduction is the penalty normally paid to gain fault tolerance capabilities. As well, the added mechanisms of synchronization and processor coordination require a careful design and verification effort. Although publications of previously developed fault tolerant systems such as SIFT and FTMP give indication of simulation work conducted in the early stages of development to test the functional concepts of each system, no simulation can provide comparisons between the performance of the system described here and the systems reported in the open literature.

2) The simulation approach to hardware and operating system software design proved to be useful in the course of this work. Several times it has been necessary to add a new feature to the guardian initialization mechanism, change the interconnection of some hardware modules or complete the description of a bus interface which the simulation proved to be inaccurate. All these changes would have been extremely time consuming and costly if they had to be done on a real prototype hardware. The inherent flexibility of the simulation approach allows us to evaluate several design alternatives of a hardware system in a short time. This

allows the designer, when he finally commits himself to implementing a certain design, to rule out early conceived options that proved to be incorrect or inefficient through the simulation work.

3) Concerning the adequacy of N.mPc as a design tool for multiprocessor systems several points can be made:

- In the course of this work the CAE tool N.mPc clearly proved to be useful as a hardware design and simulation tool. A system could be simulated and tested in a relatively short period of time.
- Frequent design changes to the simulated multiprocessor architecture showed the flexibility of N.mPc as a hardware design and simulation tool.
- On the other hand, several limitations, mentioned earlier in this summary, to N.mPc have been identified. They tend to severely reduce N.mPc's utility in the simulation and testing of fully integrated multiprocessor systems.
- A new version of N.mPc, called N.2, has been introduced by the vendor and will be available on the VAX/VMS environment. The new version N.2 incorporates a few features aiming towards making the original N.mPc more powerful. The enhancements made to N.mPc will offer only marginal improvements in the limitations of the package with respect to the simulation of fully integrated systems. It is felt that substantial changes in the structure of the package will be needed to make it applicable to

a true top-down design approach. In view of future simulation work it is expected that the next generation of CAE tools will be endowed with top-down design and simulation features to allow the designer to follow a methodology in which he can test the lower level modules of the operating system down to register transfer level of details. Higher level modules can then be simulated and tested with the already tested lower levels replaced by macro instructions or functional blocks. This will enhance the simulation performance by several orders of magnitude over what is currently attainable by a tool like N.mPc. It will then make it possible in practice to simulate a sophisticated system such as the fault tolerant architecture reported here in its full fledged configuration in a reasonable period of time and using modest computing resources.

The fault tolerant architecture simulated in this work is useful in many practical applications requiring continuous unattended operations. In addition to spacecraft on-board processing, these applications include computer communication switching gear, nuclear plant monitoring and processing applications, data collection and distribution in environmental and resource management applications and in on-site processing of remote sensing data.

TABLE OF CONTENTS

VOLUME 1

VOLUME 2

APPENDICES

LIST OF FIGURES

1. INTRODUCTION

The simulation activity described in this report is part of an overall study which includes the design and simulation of a fault tolerant architecture for multimicroprocessors. The main focus of the overall study, however, has been on the development and utilization of computer aided engineering (CAE) design tools. One of these CAE design tools, known as N.mPc, is currently running on the VAX/VMS 11/780 system located in the Analysis and Simulation laboratory of the Communications Research Centre, Department of Communications. The N.mPc package is used as a simulation environment for the work reported here. A user manual [24] and a system manual [16] for N.mPc are currently available and should be reviewed by the reader as background material for the simulation work described in this report.

A conceptual design for the hardware architecture was completed and reported in [14]. As well, a fault tolerant operating system was developed [7] and simulated separately using the C programming language under the VAX/VMS environment. The simulation conducted here integrates certain parts of the operating system with the hardware architecture using the N.mPc simulation package. While a brief review of both the hardware architecture and the operating system is provided in this report, it is essential for the reader to consult references [7] and [14] in order to gain full appreciation of the material presented in the following sections.

This study was conducted for the Communications Research Centre under a DSS contract.

1.1. Simulation Objectives

The main objectives of the simulation conducted here are:

1. To test the fault tolerant features of the hardware and to examine in detail their behaviour when faults or errors take place.
2. To provide a test bed which will make it possible to vary the structure of the fault detection and recovery components and to compare the effectiveness and performance of various alternatives.
3. To test the interfaces between the low level layers of the operating system and the hardware and verify the completeness of the description of these interfaces.

While the ultimate objective of the simulation is to integrate fully the hardware architecture and the fault tolerant operating system, the work reported here includes a partially integrated system since only certain modules of the operating system have been integrated with the fully simulated hardware. Limitations of time and resources plus a number of technical reasons (described in section 4.1 and in section 5) have constrained the current effort towards full integration.

The simulation approach to hardware design was chosen because it offers considerable benefits in the early stages of the design process. Traditionally, microcomputer based products are designed according to the following steps:

1. The necessary hardware components are built. This usually includes the microprocessor itself as well as other peripheral components.
2. Software is written for the target machine.
3. Software and hardware components are integrated and tested. Very frequently, the software is produced on a host machine using a cross development package, if available.

The development process usually involves many time consuming and costly iterations. A CAE tool such as N.mPc improves the situation by providing a simulation environment which is suitable for testing many design alternatives in a short period of time. The implications of using N.mPc are as follows:

1. It is no longer necessary to build the hardware components at the beginning of the design work. Instead, N.mPc provides what amounts to a micro-programmable, register transfer level machine which can be programmed to emulate the target hardware completely. In other words, a designer working on a VAX host, for example, could create a VAX executable program which, when run, would emulate the target hardware.
2. N.mPc provides a totally programmable cross development package for the software to be written in assembly language. The work documented in this report also uses an enhanced software development environment permitting to write software in the C high level language.
3. The rationale for using a tool such as N.mPc is that programmability implies flexibility. Given that a base exists, i.e. most of the hardware emulation is available as well as the cross development package, a designer can alter the design parameters with ease and test various alternatives without committing to any hardware choice. The advantages and disadvantages of the above mentioned design methodology are also discussed in [16].

1.2. Report Structure

Section 2 of this report describes briefly the conceptual design of the fault tolerant multiprocessor architecture and provides an overview of the simulation of the hardware using N.mPc. An overview of the operating system is also included in section 2. The testing of the simulated fault tolerant hardware architecture is documented in section 3. Section 4 reports the testing of the hardware dependent routines of the fault tolerant operating system. It is shown that these routines constitute an interface between the fault tolerant operating system and the hardware. Conclusions as well as a summary of achievements are found in section 5. Section 6 contains recommendations for future simulation work.

All appendices are contained in volume 2 of this report.

2. SIMULATION IMPLEMENTATION DESCRIPTION

2.1. Simulation Environment

The N.mPc package, running on a VAX 11/780 (located at CRC) and the VMS operating system, provided the simulation environment for the work presented here. A brief description of N.mPc is presented in the following.

N.mPc consists of six components used to describe the hardware behaviour of a target system and to execute the simulation of that system. Figure 2-1 illustrates the components of N.mPc and their interaction.

The Meta-micro assembler and the linking loader are used to generate the software which is to be executed by the simulated hardware components if these are programmable. Both are driven by a description of the instruction set of a target machine and can be made to generate code for either vertically or horizontally programmed machines [16]. The linking loader produces code which is executed by a simulated processor or by an actual machine. The ISP′ compiler is used to produce simulation modules for individual processors and other hardware components of a system. The input language of the compiler is the ISP′ language which allows the specification of states for the implementation of processor registers and flags, memories for the simulation of memory, and ports which allow input to and output from simulated hardware.

The N.mPc ecologist and a simulated memory processor link the ISP′ processor modules with the linking loader outputs in order to form complete simulations. A run-time package is used to execute a simulation and to allow extensive user interaction with the simulation.

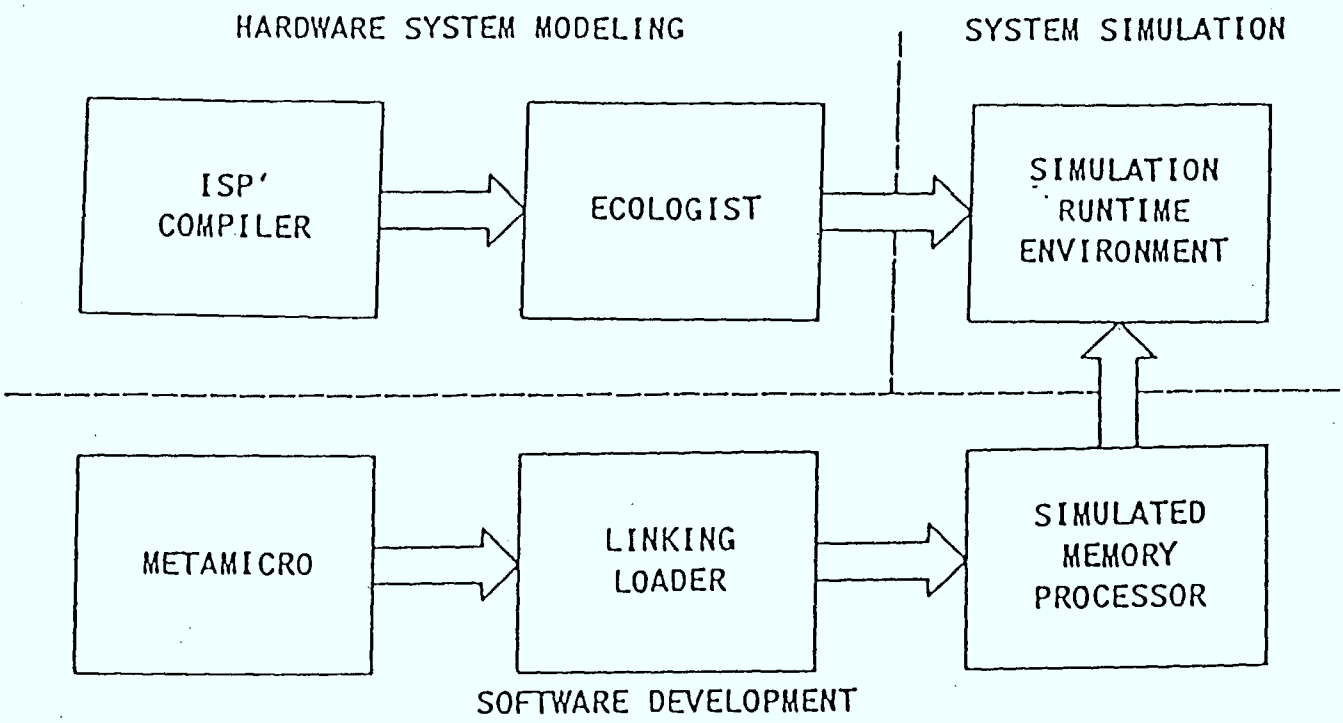


Figure 2-1: Elements of the N.mPc System

Other reports directly related to this work describe the conceptual design of a fault tolerant multiprocessor operating system [7] and of a fault tolerant architecture for multiprocessor systems [14]. Further information on N.mPc and work in the area done at Intellitech is given by references [8, 16, 20, 23, 24] as well as the original N.mPc documentation listed as references [1-6, 9-13, 15, 18, 21, 22].

N.mPc is used to simulate the fault tolerant multiprocessor architecture by first producing descriptions of all necessary hardware modules in N.mPc's hardware description language ISP'. The next step is to describe an interconnection scheme for the simulated hardware components of the multiprocessor architecture.

Following the hardware description, some test software for the programmable hardware modules (descriptions of the Intel 8086 microprocessor) has to be developed. Simulated hardware and test software for the microprocessors are integrated under N.mPc to form an executable simulation that allows testing of the fault tolerant multiprocessor architecture. The behaviour of the multiprocessor system in response to certain failures can also be simulated.

2.2. Fault Tolerant Multiprocessor System Overview

This section presents an overview of a fault tolerant multi-microprocessor system for spacecraft on-board processing applications. The fault tolerant multiprocessor architecture can also be used in general applications requiring a high degree of reliability over a specific processor life cycle.

The approach chosen for the design of the fault tolerant multiprocessor system is novel in the sense that fault tolerant features and supporting mechanisms are embedded in both the hardware architecture and the operating system software. The hardware has multiple redundant components that are controlled by fault detection mechanisms in order to prevent the propagation of errors from the faulty component to the remainder of the system. The operating system software contains all the intelligence needed to detect errors, identify their sources, take the necessary action to remove faulty units, reallocate the processing tasks and reconfigure the system to adapt to the new operational state.

Figure 2-2 gives an overview of the fault tolerant multiprocessor architecture which is aimed primarily for spacecraft on-board processing applications. The central processors (CPs) share the computational load of the satellite and are the highest onboard processing authority. They are built as a fault tolerant structure and employ redundancy to ensure reliability. The peripheral processors usually operate alone and can be considered as slave processors controlled by the central processors. Another set of processors, the interface processors (IPs), provides an interface between the loosely coupled central processors and the tightly coupled peripheral network. The "gate complexes" come in groups of four since four redundant buses are used in the interface between central and peripheral processors. They are redundant components controlled by fault detection mechanisms ("guardians") in order to isolate the sources of error and prevent the proliferation of these errors from the faulty component to the remainder of the system.

Figure 2-3 shows the different layers of the fault tolerant operating system. At the lowest level is the hardware itself. The

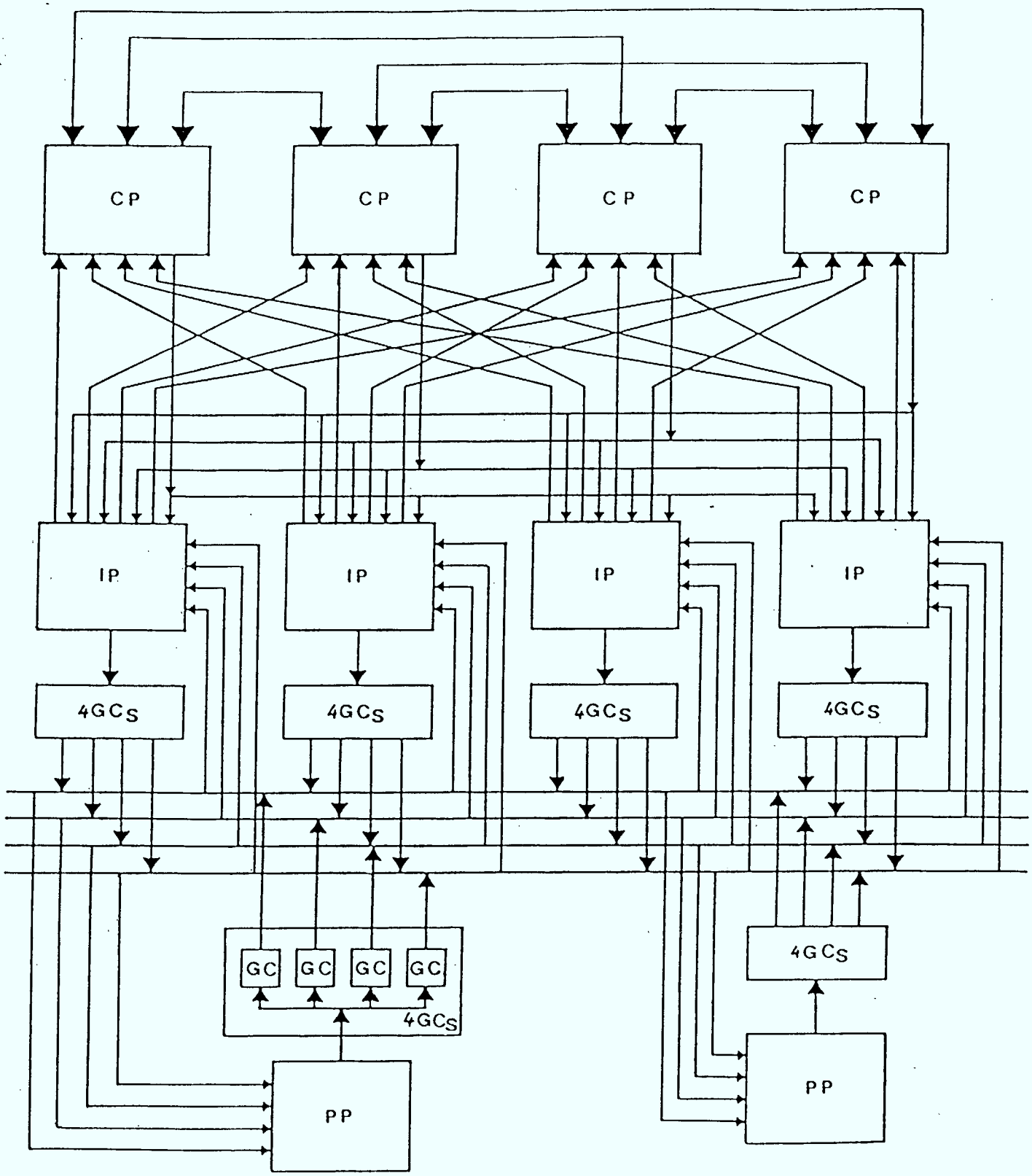


Figure 2-2: Overview of the Fault Tolerant Hardware System

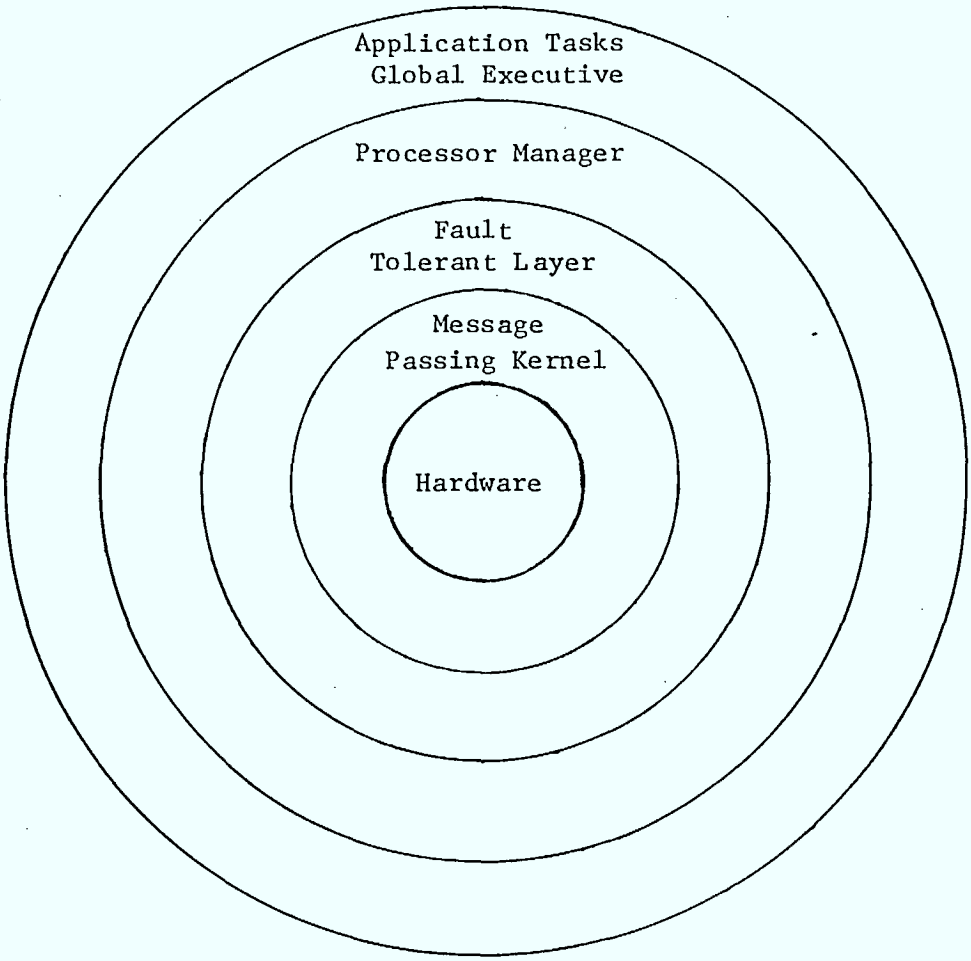


Figure 2-3 : The Fault Tolerant Operating System

layer above that, the message passing kernel, is a simple message passing system without any fault tolerant features but providing the basic operating system functions. The layer above the kernel implements the virtual machine visible to the application tasks and fault tolerant facilities such as error detection, message traffic control, voting, buffer management, task scheduling and reconfiguration. The next higher layer, the processor manager, does not incorporate fault tolerance mechanisms but directs and coordinates most of the fault handling mechanisms of the layer below itself. Sitting on top of the processor manager layer are the global fault tolerant facilities (the global executive), which are responsible for synchronization and reconfiguration. The global executive handles the assignment of tasks to processors, reconfiguration of the system around failed parts, isolation of faulty parts, system updates, etc. To ensure that the global executive, the highest on-board authority of the satellite processing system, operates without faults, it is implemented in multiple copies and is executed on multiple processors.

Examples of the hardware faults which can be tolerated (ie. recovered from) by the full fault tolerant multiprocessor system include:

- Malfunction of a central processor.
- Malfunction of an interface processor.
- Failure of one of the redundant buses between interface and peripheral processors.
- A failing peripheral processor (device) is simply cut off from the redundant bus by its guardians.
- Failure of a gate complex (guardians, gates).
- Errors in received data due to hardware failures.

2.3. Implementation of the Fault Tolerant Hardware Simulation

This section describes the implementation of each hardware module needed for the simulation of the fault tolerant multiprocessor architecture. The interconnection of all hardware modules, resulting in a full fledged fault tolerant multiprocessor hardware architecture, is then shown.

2.3.1. The Central Control System

2.3.1.1. Processor Module

The central control system consists of a number of loosely coupled, fully interconnected processor modules as shown in Figure 2-4. Each processor module consists of an 8086 CPU, a memory, a timer, 3 ports for communication between central processors and 5 ports for communication to and from all interface processors. Figure 2-5 shows a central processor module. The 8086 CPU and the memory(16k, RAM) have been discussed in detail in [20]; Figures 2-6 and 2-7 show their ports and interconnection within the system. The timer shown in Figure 2-8 is able, when adressed, to stop the CPU for a duration of time that can be specified.

2.3.1.2. Processor Network

In this section the ways in which the various modules of the fault tolerant system communicate will be presented. Communication is effected through a number of communication ports. Depending on their use communication ports may differ but they all consist of the same basic units, namely a transmitter and a receiver.

Communication ports intended to connect central processor modules to other central processor modules consist of both a transmitter and a

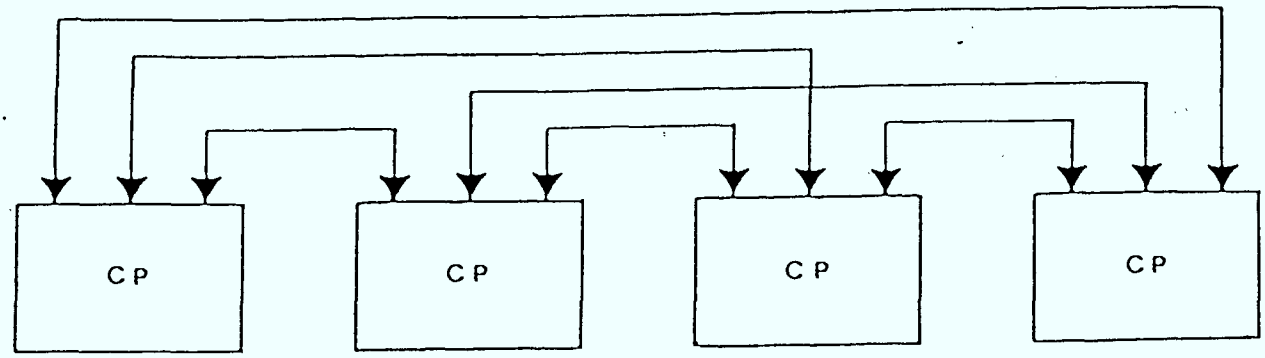
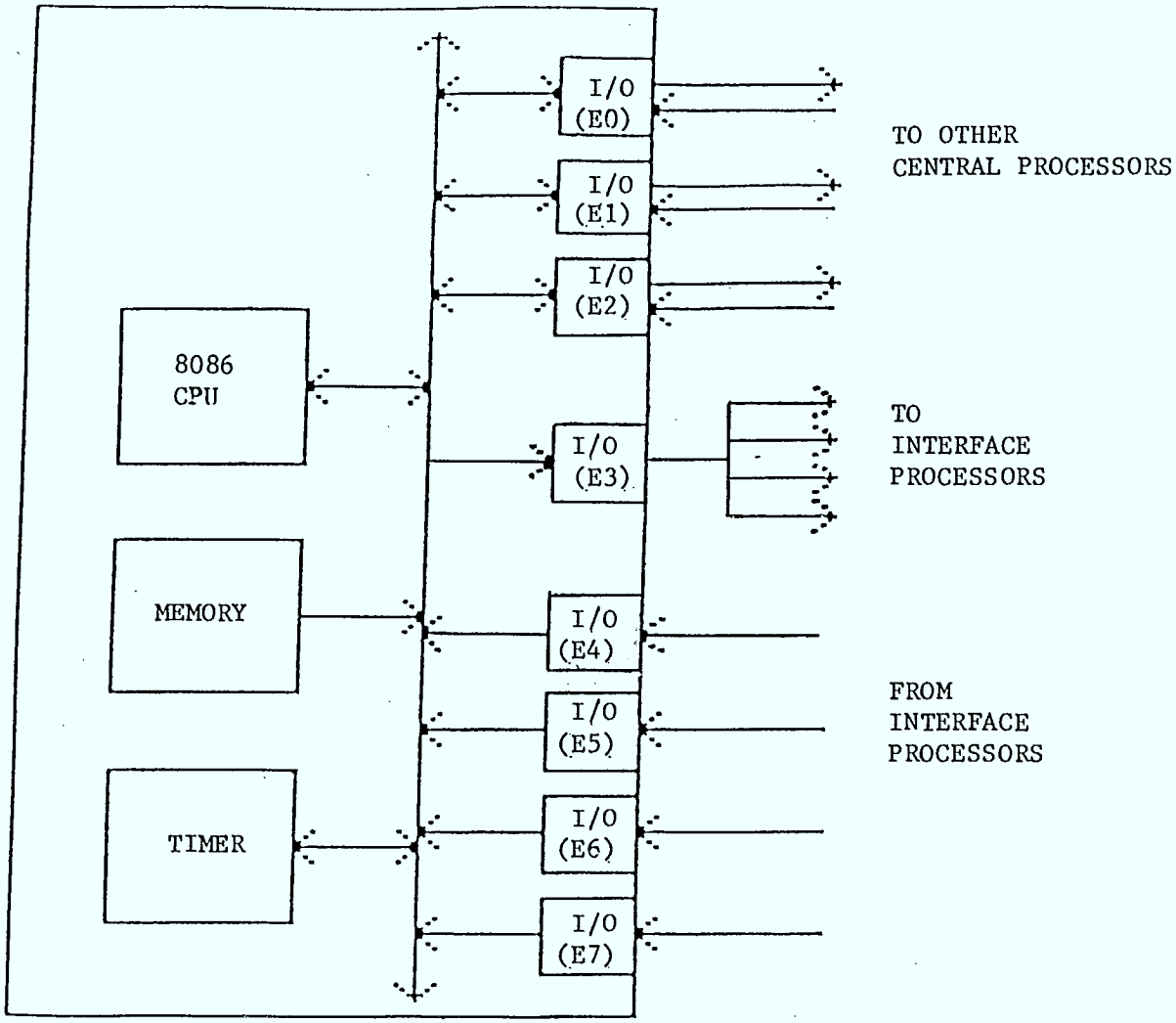
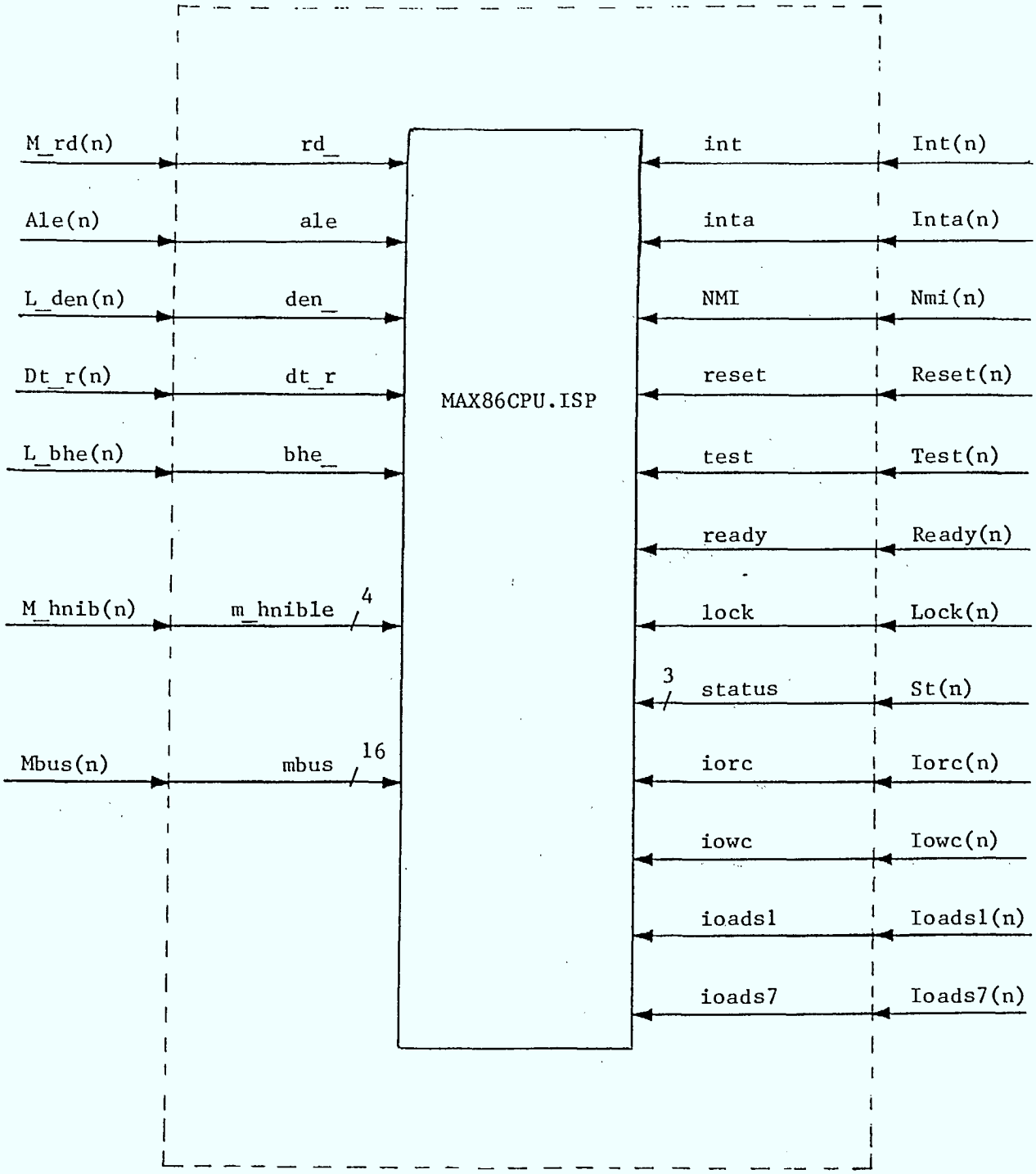


Figure 2-4: The Central Control System



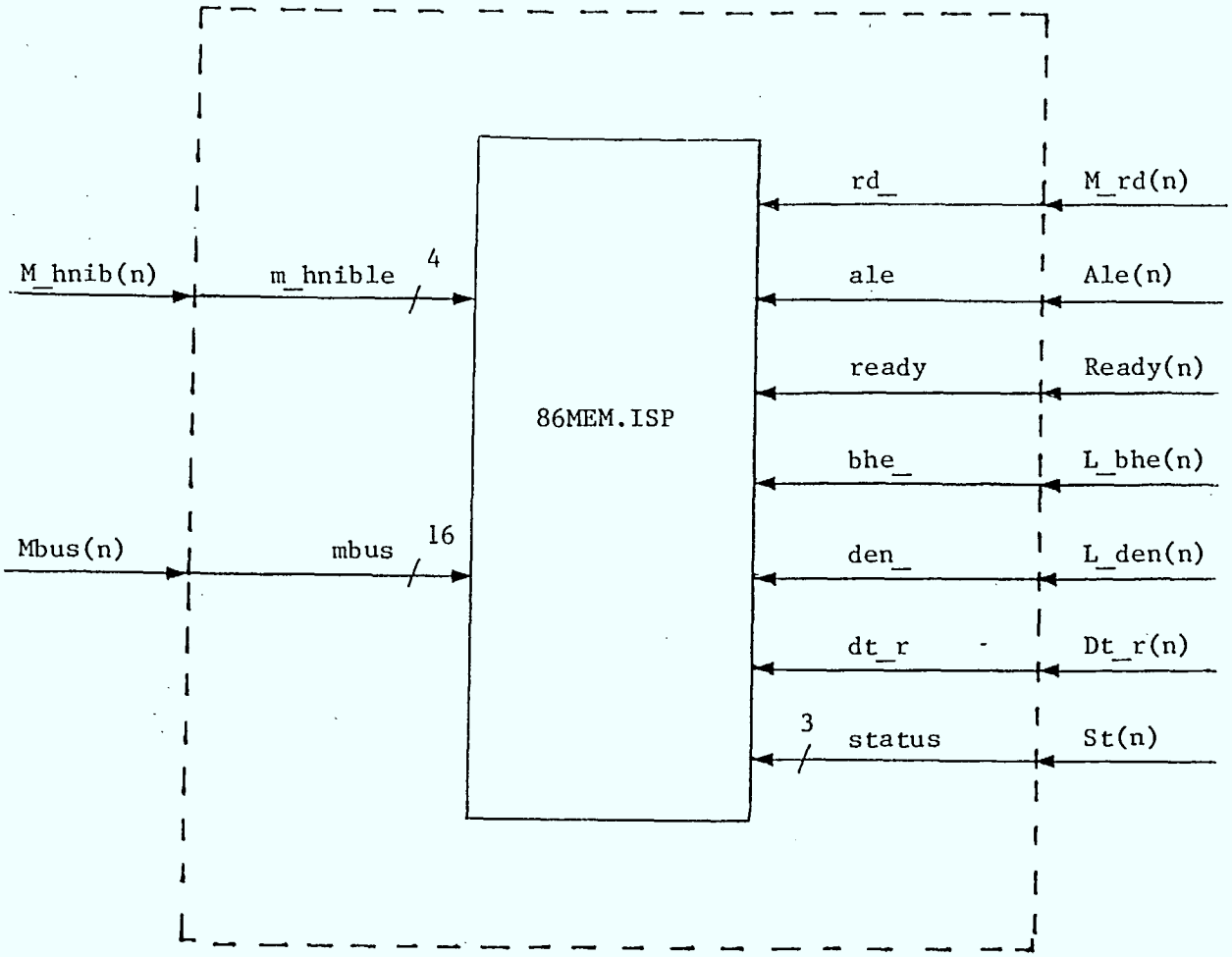
NB: The I/O addresses used to access an IO port or device are indicated in brackets

Figure 2-5: Central Processor Module



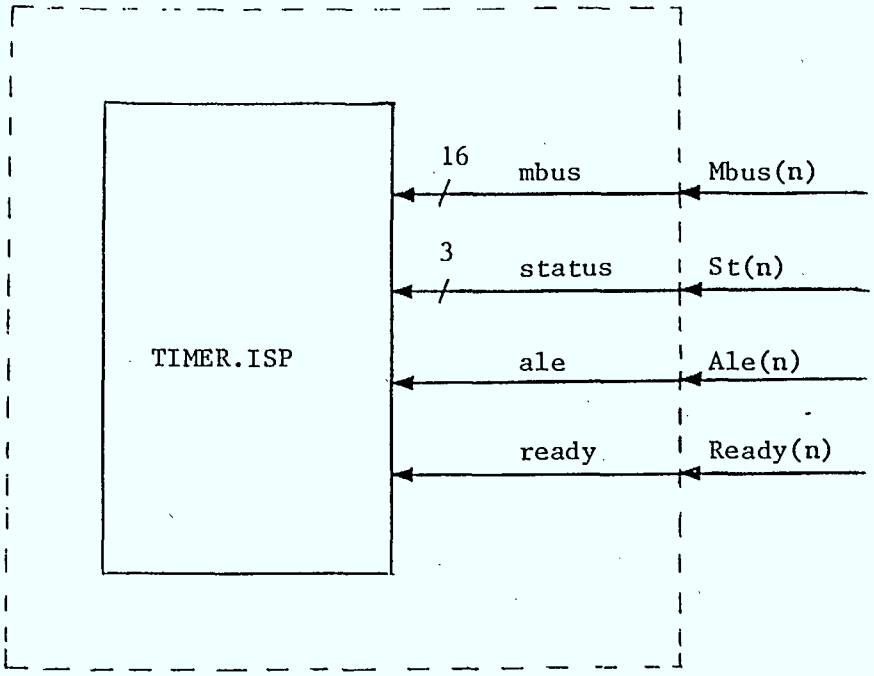
NB: The number n identifies a certain 8086 CPU.

Figure 2-6: The Intel 8086 CPU Module



NB: The number (n) depends on the processor to which the memory is connected

Figure 2-7: The Memory Module



NB: The number n depends on the processor to which the timer is connected

Figure 2-8: The Timer Module

receiver. Although united in one module(trx.isp) transmitter and receiver operate independently of each other. Each central processor module has a dedicated port connected to every other processor module so that a fully interconnected system is formed.

In addition the central processor modules have a single transmitter and a number of receivers that handle the communication with the interface processors. This special interface is necessary because of the synchronous nature of the interface processors.

The block diagram of a transmitter is shown in Figure 2-9. The transmitter consists of a FIFO queue and a serializer that converts the parallel data into serial data ready for transmission. The serializer can be thought of as the transmitter part of a UART. The FIFO queue serves to offload the processor from continuously checking the transmitter and to give the transmitter the capability of autonomous operation for a certain length of time. The size of the FIFO is chosen so that the transmitter can operate without any attention from the processor for the duration of the time slice of a task. In other words the transmitter requires the attention of the processor only during a context switch. This greatly simplifies the design and implementation of the operating system.

For example a processor module that wishes to send a message to another module can write the message into the transmitter FIFO and then proceed with its other tasks while the transmitter is sending the message. The processor sees the transmitter as a pair of I/O ports. One is a data port and the other is a command port. Data written into a data port are placed in the FIFO and transmitted. The command port is addressed by using the normal data port I/O addresses (E0-E7)

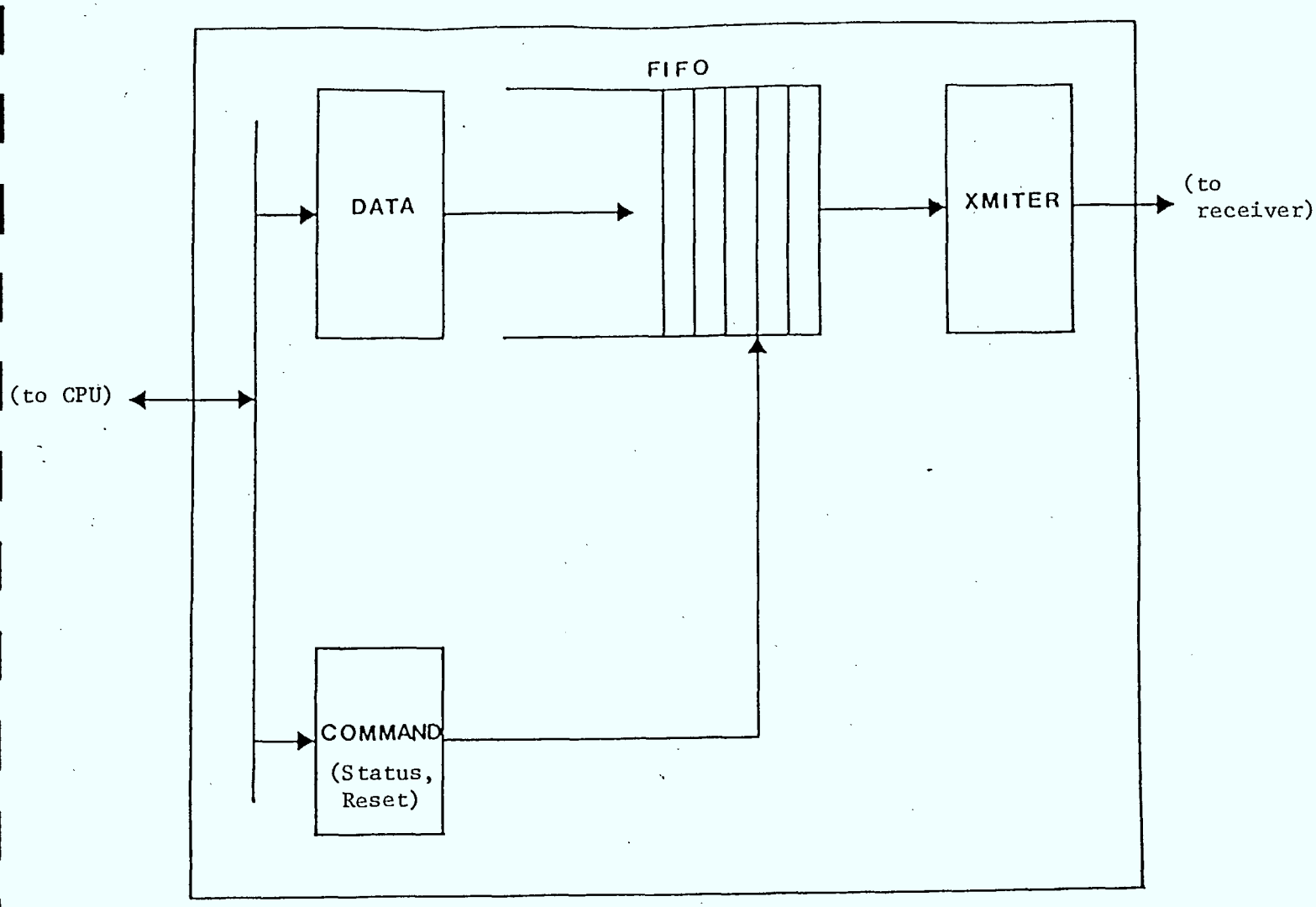


Figure 2-9: Block Diagram of a Transmitter

increased by eight (E8-EF). The command port enables the processor to control the transmitter and to check its internal status. The main interest here is in being able to reset the transmitter and to check if the queue is full.

The structure of the receiver, shown in Figure 2-10 is complementary to that of the transmitter. The receiver consists of a deserializer and a FIFO. The deserializer is actually the receiver part of the UART. When a byte of data has been collected it is placed in the FIFO for pickup by the processor. Like the transmitter, the receiver appears to the processor as a pair of ports. One port is a data port and the other a command port. A read from the data port will return the first byte in the FIFO. If the FIFO is empty the result is undefined. The command port enables the processor to control the receiver and check its status. Presently we need to be able to reset the receiver and check if the queue is empty or if it has overflowed.

In the N.mPc simulation the serializer and deserializer have been omitted. Instead data is sent one byte at a time. Also a handshaking protocol has been included to ensure reliable communication through interconnections shown in Figure 2-11. This simplifies the simulation without affecting its validity. Appropriate delays have been incorporated in the simulation of the transmitter. They simulate the delays caused by serial transmission of data. For communication between central processors and interface processors, as well as for inter central processor communication, receiver and transmitter have been united in one module called "transceiver" (trx.isp). Figure 2-12 shows the transceiver which can be used for two way communications or also just as a receiver or transmitter. The ISP´ description files for the

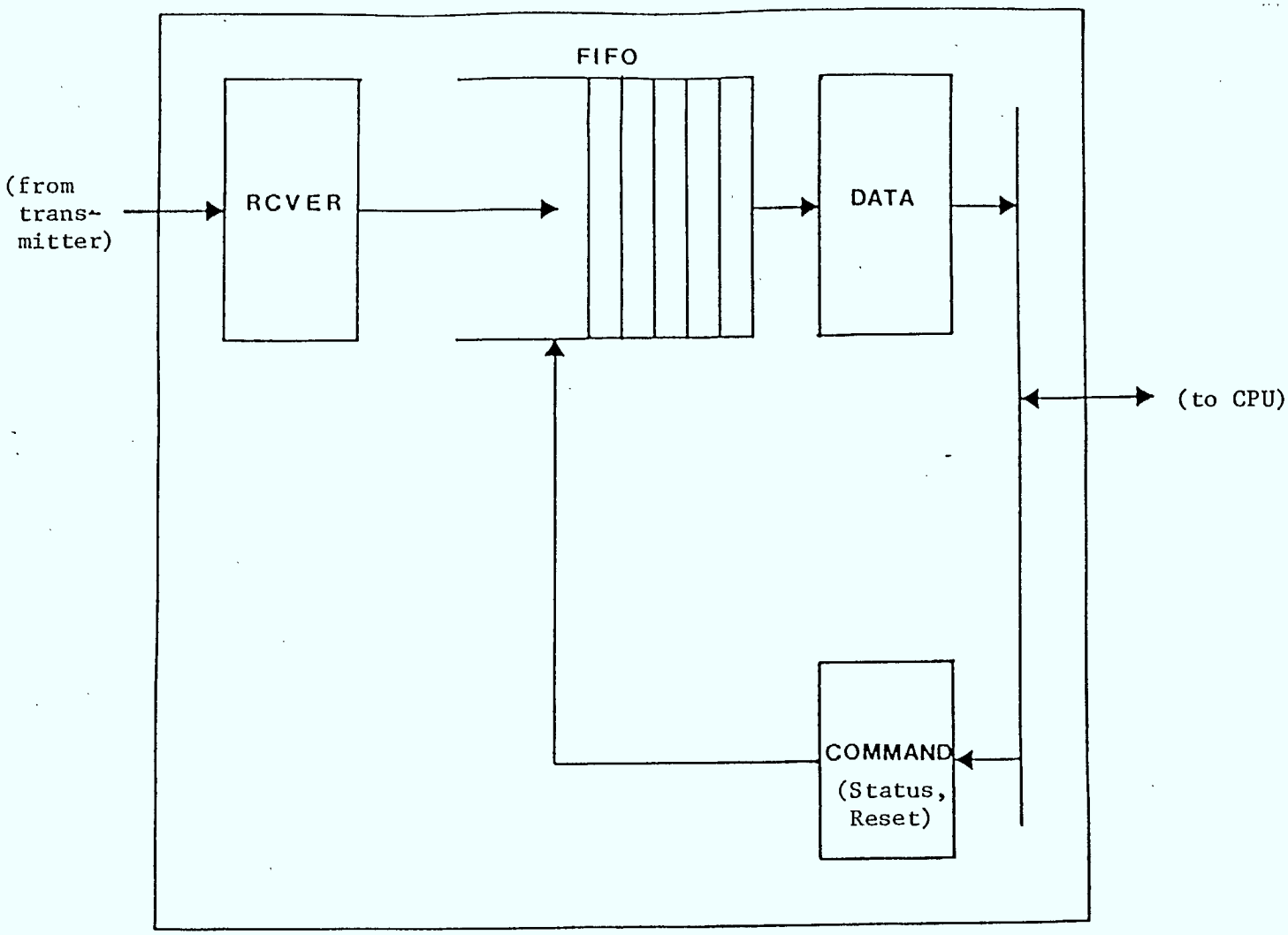


Figure 2-10: Block Diagram of a Receiver

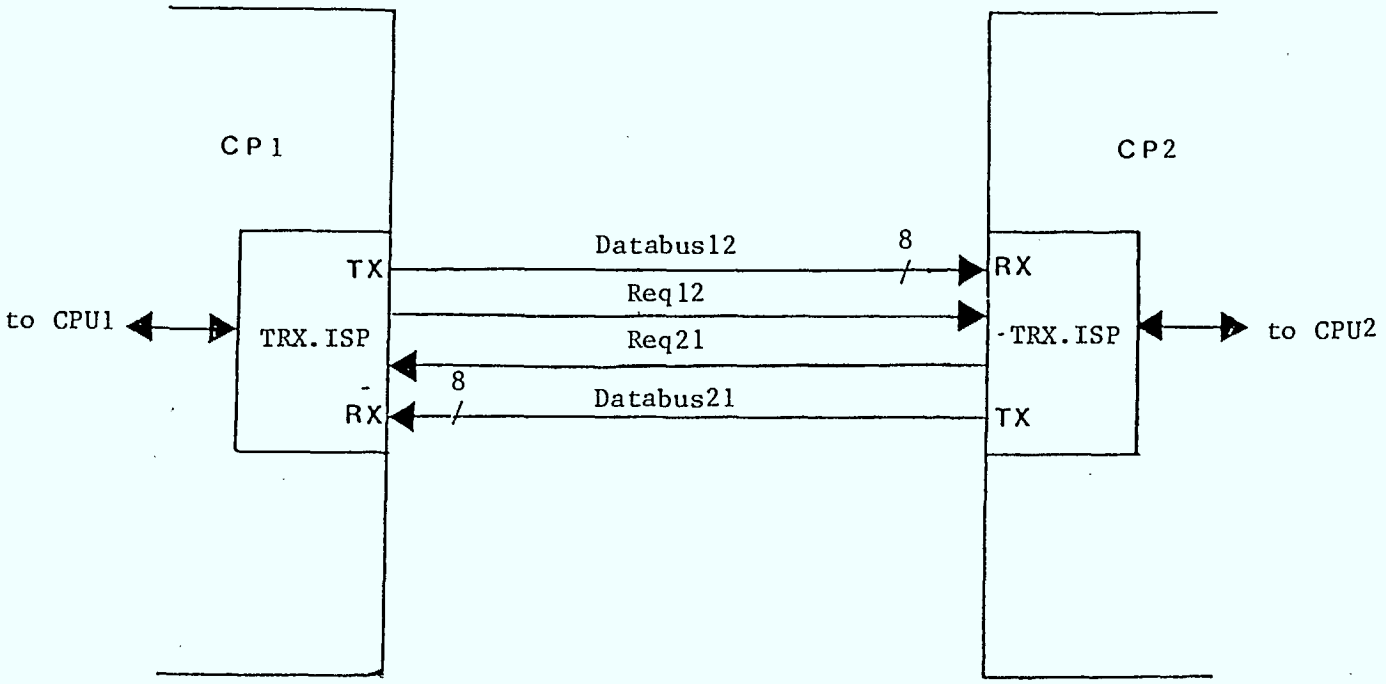
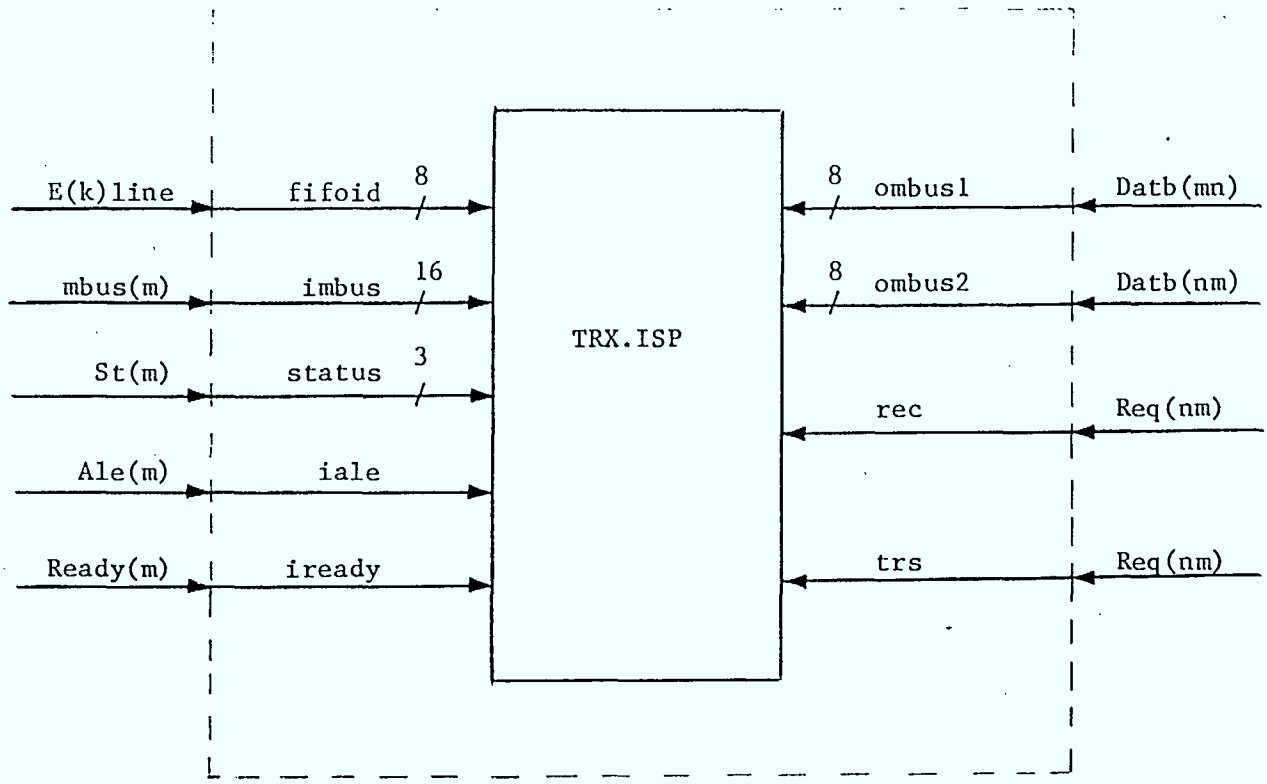


Figure 2-11: Block Diagram of Central Processor Interconnection



NB: (k), (m) and (n) are numbers which depend on what other modules are connected to a transceiver

Figure 2-12: The Transceiver Module

transceiver (transmitter-receiver) and the timer are given in Appendix A.1.

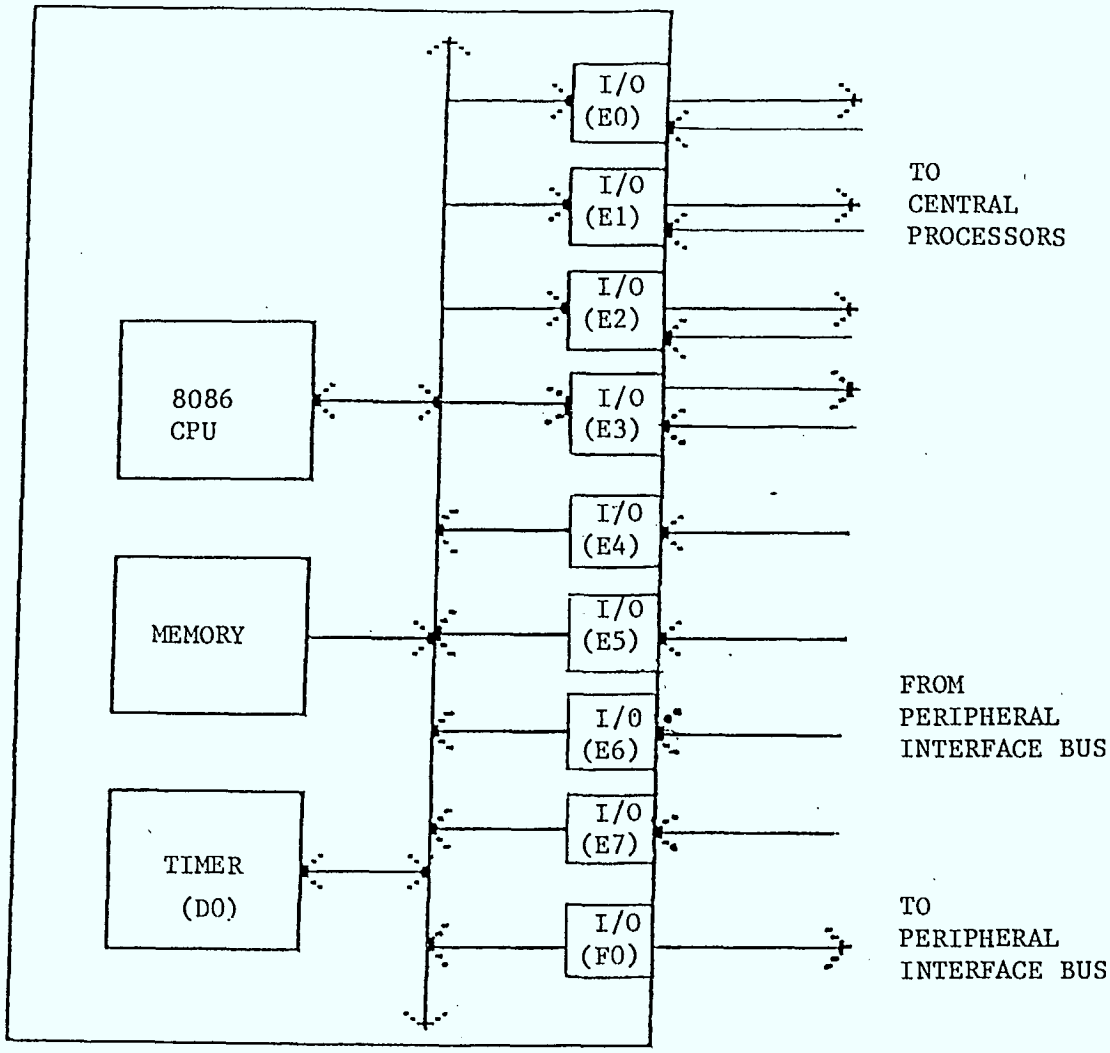
2.3.2. The Peripheral Interface System

2.3.2.1. The Interface Processor

In [14] it was determined that the peripheral interface system must provide an interconnection between the loosely coupled central processors and the tightly coupled redundant bus (also called "Peripheral Interface Bus", PIB) without introducing a single point of failure. To accomplish this a number of interface processor modules and fault tolerant hardware interfaces ("gate complexes") are used. The overview of the multiprocessor architecture in Figure 2-2 depicts the situation.

The interface processor modules are no different from the central processor modules in so far as they both contain the same 8086 CPU, memory and timer. However, the interface processor modules do have a different set of communication ports. Figure 2-13 shows the components that are part of an interface processor module.

The communication ports on the interface processors that connect them to the processor modules of the central system are identical to the ports that interconnect the processor modules to each other. However the ports that connect the central processor modules to the interface processors and the ports that connect the interface processors and the peripheral processors to the redundant bus are different. These ports consist of a single transmitter and a number of receivers. In both cases the transmitter broadcasts to all concerned units (either interface processors or buses). However there is a dedicated receiver for each distinct unit that is connected.



NB: The I/O addresses used to access an I/O port or device are indicated in brackets

Figure 2-13: The Interface Processor Module

In the case of the central processor modules the single transmitter ensures proper communication between the loosely coupled central system and the tightly coupled interface processors. In other words it ensures that all the interface processors will receive their copies of the message simultaneously. This is essential for the interface processors to remain in perfect synchronization. Even if the central processors were synchronous the time difference between sending the same message to different interface processors would be enough to throw them out of synchronization. In the case of the interface processors the processors need not know to which of the redundant buses they are connected. So they simply transmit through a single transmitter and the gate complexes ensure that only the proper bus is driven.

2.3.2.2. The Peripheral Processor

The peripheral processor module is virtually identical to the interface processor module shown in Figure 2-13. The only difference between the two is that a peripheral processor is not connected to the central processors and therefore does not need the four I/O ports used for this purpose in the interface processors.

2.3.2.3. The Peripheral Network

An interface processor has one communication port for each processor in the central control system. No special care need to be taken for these ports since the central control system is asynchronous. Actually these ports are identical to the ports the central processor modules use to communicate with each other. Aside from these ports the interface processor also has a single transmitter and four receivers. The function of the transmitter is to drive one of peripheral interface buses whereas the four receivers "listen" to the four peripheral

interface buses. Figure 2-14 shows the connection of interface processors to the peripheral interface bus.

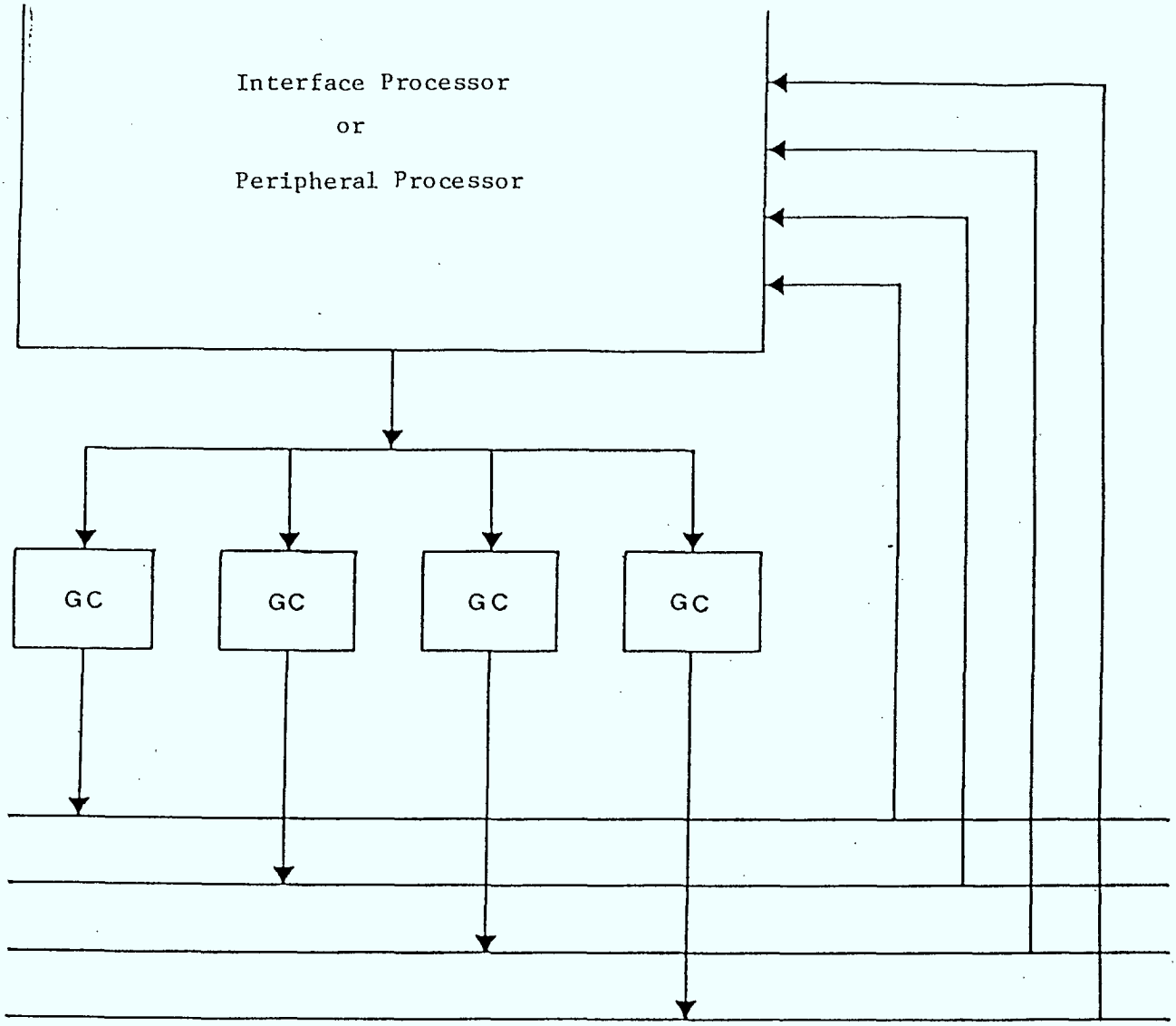
Actually any interface processor only drives one of the buses that comprise the peripheral (redundant) bus. However this is taken care of by the gate complexes and is of no concern to the interface processor. An interface processor also has a number of receivers. The function of these receivers is to pick up bit streams from the peripheral bus. Each receiver picks up data bits from only one of the buses. Thus by comparing the bit stream received from different receivers, it will be possible for the interface processor to detect bus failures.

Two of the interface processor elements, the transmitter and the receivers for the communication via redundant bus, have not previously been presented in detail. The ports and interconnection of transmitter ("gtintrfc.isp") and receiver ("bsfifo.isp") are given in Figures 2-15 and 2-16.

Both receiver and transmitter use a simple handshake protocol to receive data from or send data through the redundant bus. This protocol is different from the one used for communication between central processors. The receiver automatically puts data received from the redundant bus in a queue. Before the 8086 CPU reads from the queue it can (and should) test whether the queue is empty or not. The transmitter takes data sent to it by a CPU and automatically puts it on the redundant bus, where it is received by any "listening" receiver.

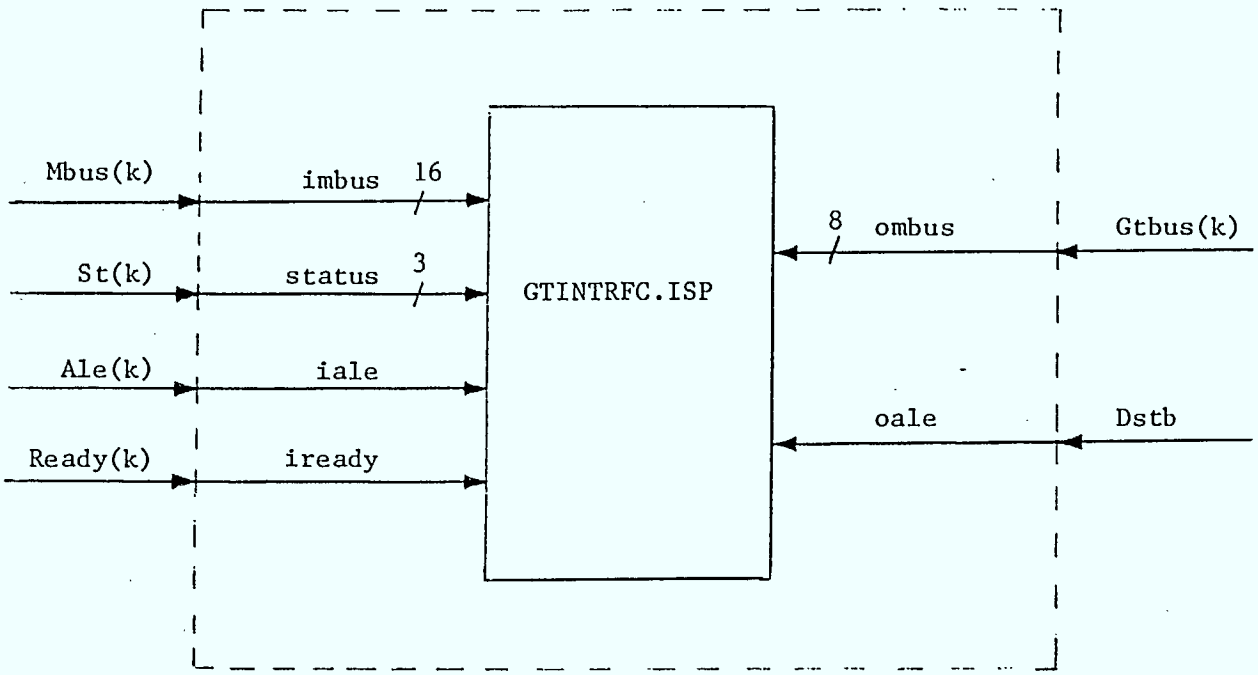
2.3.2.4. The Gate Complex

A gate along with a number of guardians form a gate complex. The function of a gate complex is to allow a processor to access a bus in a controlled and fault tolerant fashion. The block diagram of a gate



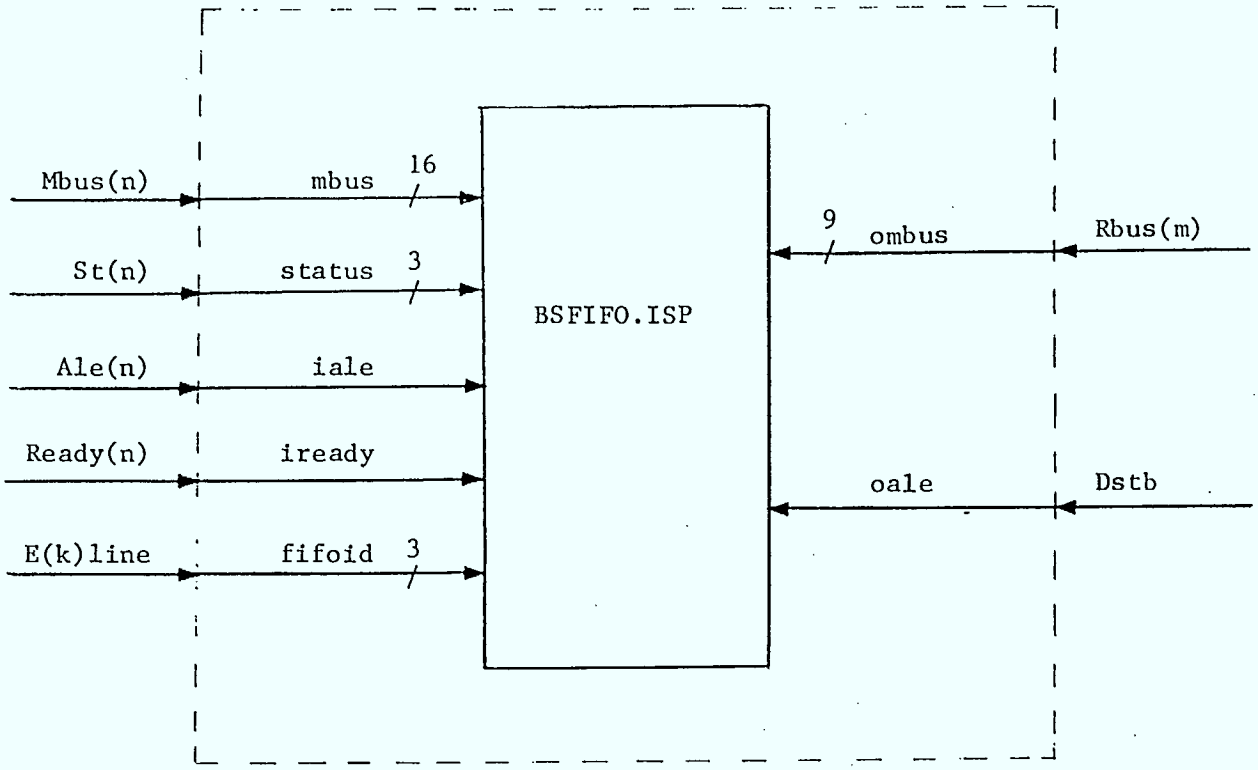
GC: Gate Complex

Figure 2-14: Connection of Processors to the Redundant Bus



NB: (k) is a number which depends on what other modules are connected to a transmitter.

Figure 2-15: The Transmitter Module



NB: (k), (n) and (m) are numbers which depend on what other modules are connected to a receiver.

Figure 2-16: The Receiver Module

complex is shown in Figure 2-17. The complex consists of a gate and three guardians. Each processor connected to the peripheral network is connected to each of the buses that form the redundant bus via a distinct gate complex. It is essential that no two gate complexes have any hardware in common so that hardware failures are as local as possible. Thus it is not possible to utilize a quad buffer chip to implement four gates. Each gate must be physically separate from any other gate.

A gate allows a processor to access the bus only if it is enabled by all its three guardians. Each guardian enables its gate only after having been switched on during the initialization or after receiving appropriate commands from the redundant buses.

A gate is a simple tri-state buffer with multiple enables. Since the peripheral buses are serial the gate only needs to control one line. A gate can be implemented simply as a combination of two standard TTL gates as shown in Figure 2-18.

For simulation purposes the peripheral buses were implemented as byte wide parallel buses. This was done to avoid the unnecessary overhead of simulating serializers. Instead the gate has become a byte wide buffer that operates much like the single bit version described above. The block diagram of this implementation is given in Figure 2-19. The other important components of a gate complex are the "guardians".

Basically a guardian is a simple finite state machine. The state diagram of a guardian used in the gate complexes of peripheral processors is given in Figure 2-20. This FSM has a number of inputs and a single output. Normally a guardian is idle with its output set so as to disable the gate. At this state the peripheral processor guardian

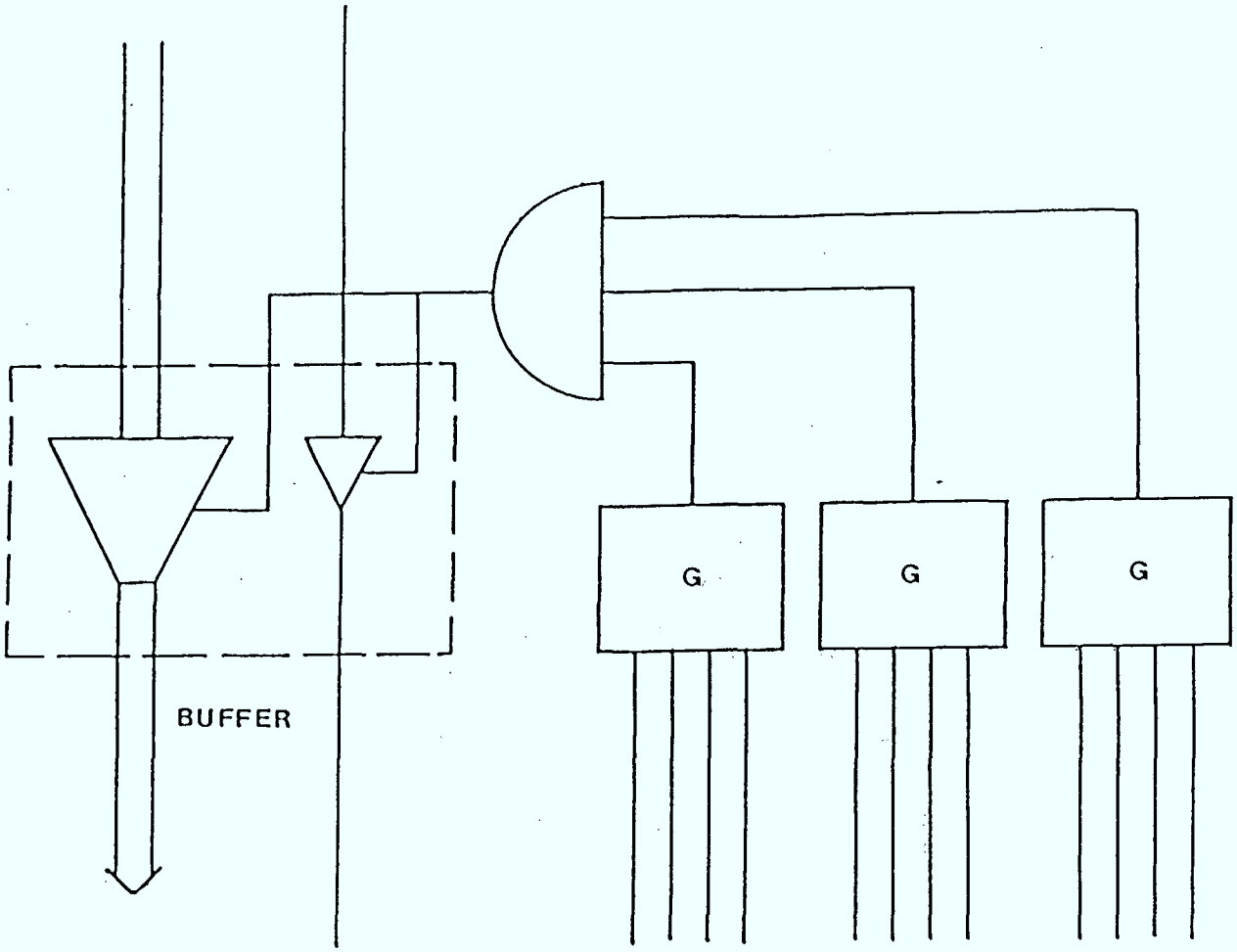


Figure 2-17: Block Diagram of a Gate Complex

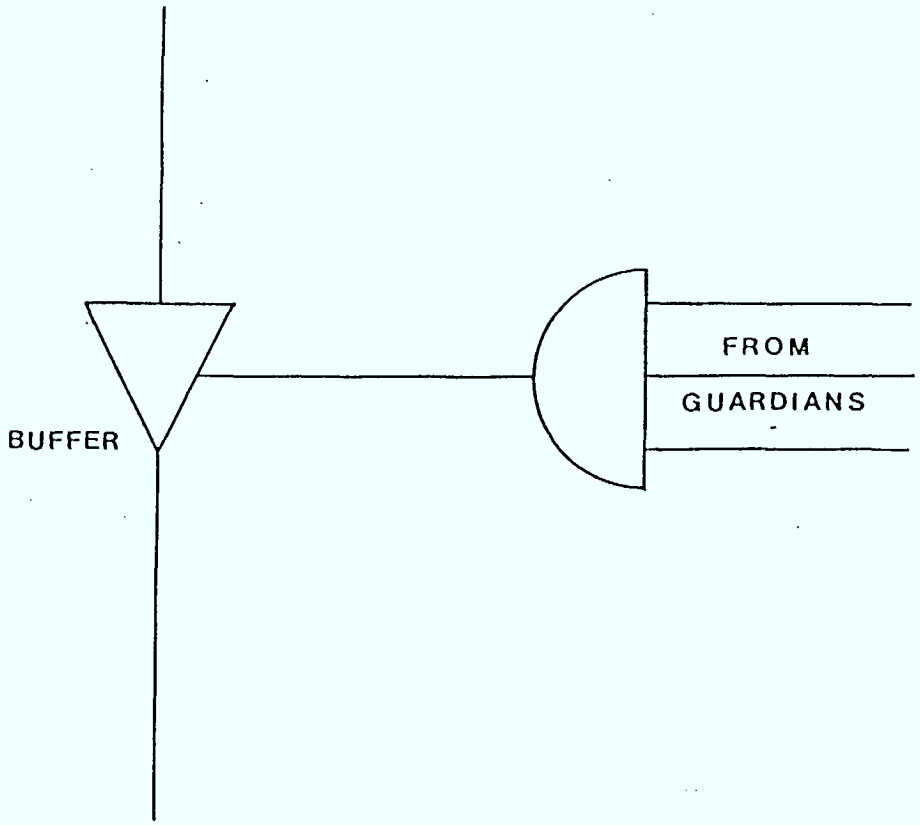


Figure 2-18: Implementation of a Gate With Agreement

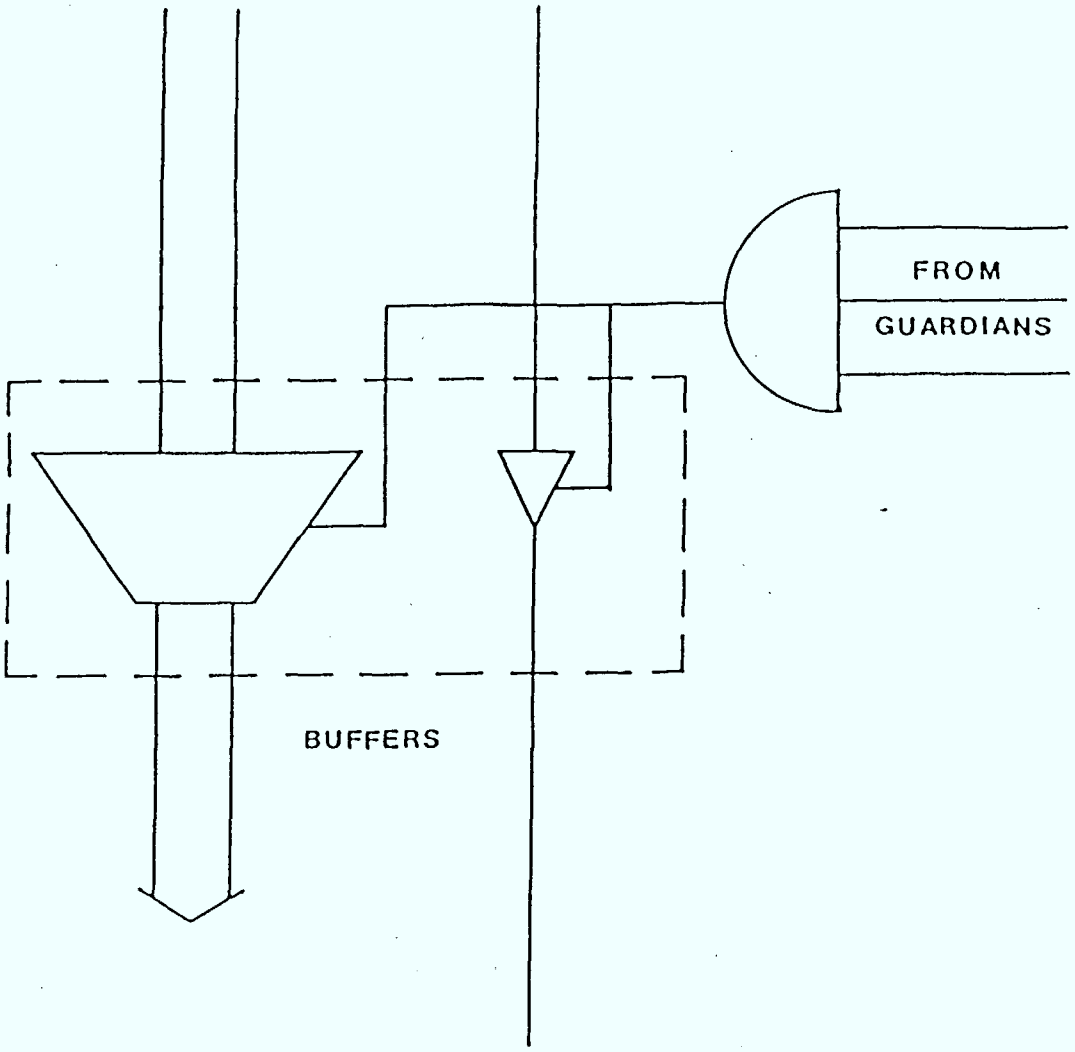


Figure 2-19: Parallel Implementation of a Gate

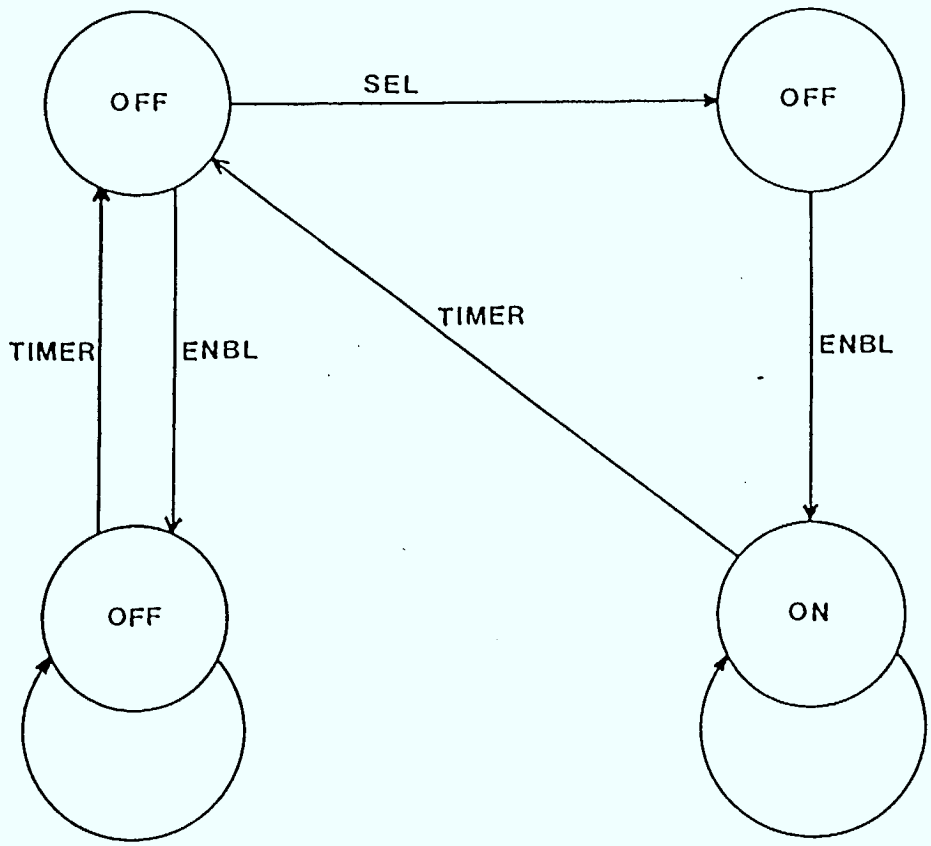


Figure 2-20: State Diagram of a Peripheral Processor Guardian

monitors the bus until it recognizes a command on it. The list of commands a peripheral processor guardian can recognize is given in Figure 2-21. Leaving the special case of a change in the triad of active buses, a peripheral processor guardian will remain idle until it recognizes either a Select or an Enable command. When a Select command is detected the guardian sets an internal flag and awaits an Enable command. As soon as the Enable command is received the guardian sets its output so as to enable the gate and leaves it enabled for a fixed period of time. At the end of this time period the guardian disables the gate and returns to its idle state. If an Enable command is detected but the guardian has not received a Select command then the guardian does not enable the gate. However it still waits for the same fixed period of time. While waiting the peripheral processor guardian ignores the bus and any information on it. Thus it is not possible for some data on the bus to be interpreted as commands and cause erroneous operation of the system.

The operation described above only refers to a guardian attached to a peripheral processor or a peripheral device. Guardians attached to the interface processors have a slightly different state diagram, as shown in Figure 2-22. Figure 2-23 shows that the commands they understand are also slightly different. Such a guardian will still ignore the bus for a fixed period of time after it detects an Enable command. However once enabled it remains enabled until explicitly being disabled. Thus an interface processor that has been given control of a bus maintains this control except for the short periods of time that it grants the bus to a peripheral for some data transfer.

Switch Bus

0	0	0	X	X	X	X	X
old bus #				new bus #			
guardian #							

Select

0	0	1	X	X	X	X	X
guardian #							

Temporary Enable

0	1	1	X	X	X	X	X

Figure 2-21: Commands for Peripheral Processor Guardians

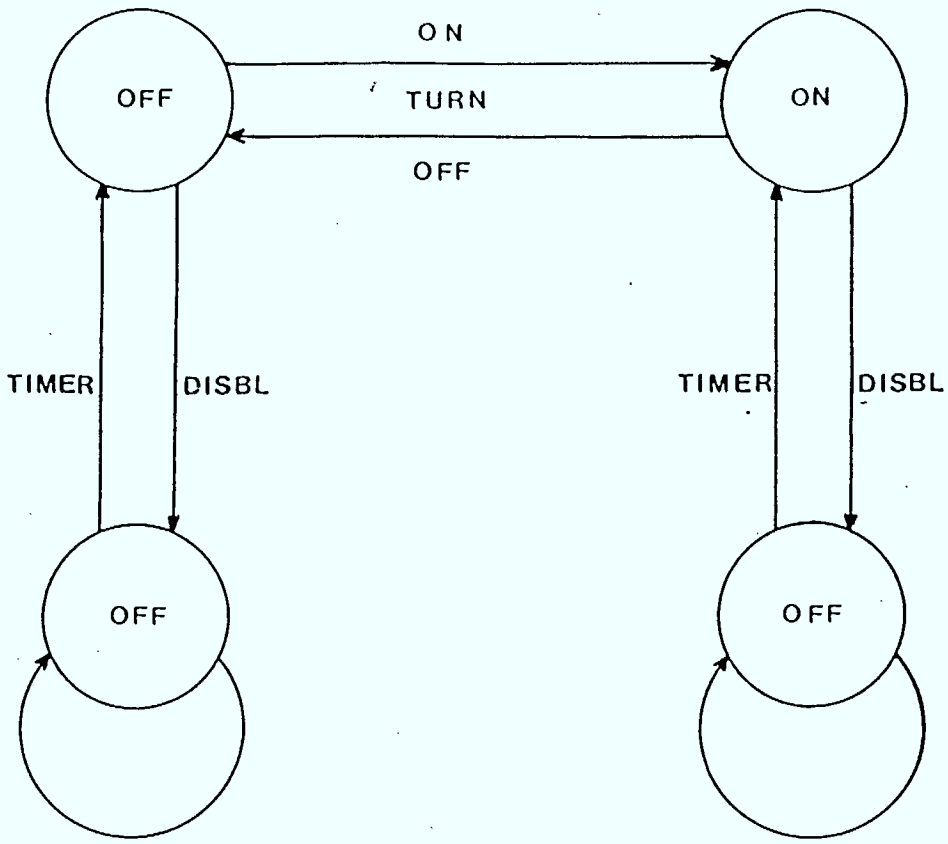


Figure 2-22 : State Diagram of Interface Processor Guardian

Switch Bus

0	0	0	X	X	X	X	X
old bus #				new bus #			
guardian #							

Turn ON

0	0	1	X	X	X	X	X
guardian #							

Turn OFF

0	1	0	X	X	X	X	X
guardian #							

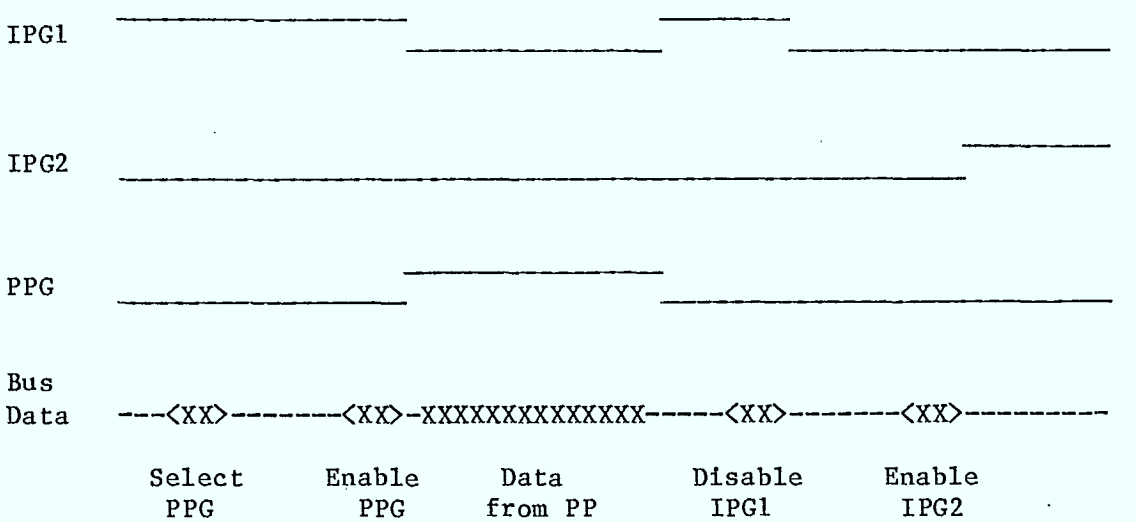
Temporary Disable

0	1	1	X	X	X	X	X

Figure 2-23: Commands for Interface Processor Guardians

Figure 2-24 shows the outputs of the two types of guardians as a function of time and of the commands received. We can observe how the guardian of a peripheral processor is normally OFF whereas the guardian of an interface processor is normally ON. When the guardian of the peripheral processor is selected its output does not change. However, when the Enable command is sent its output is activated and remains so for a fixed period of time. During this time all other guardians in the system disable their outputs.

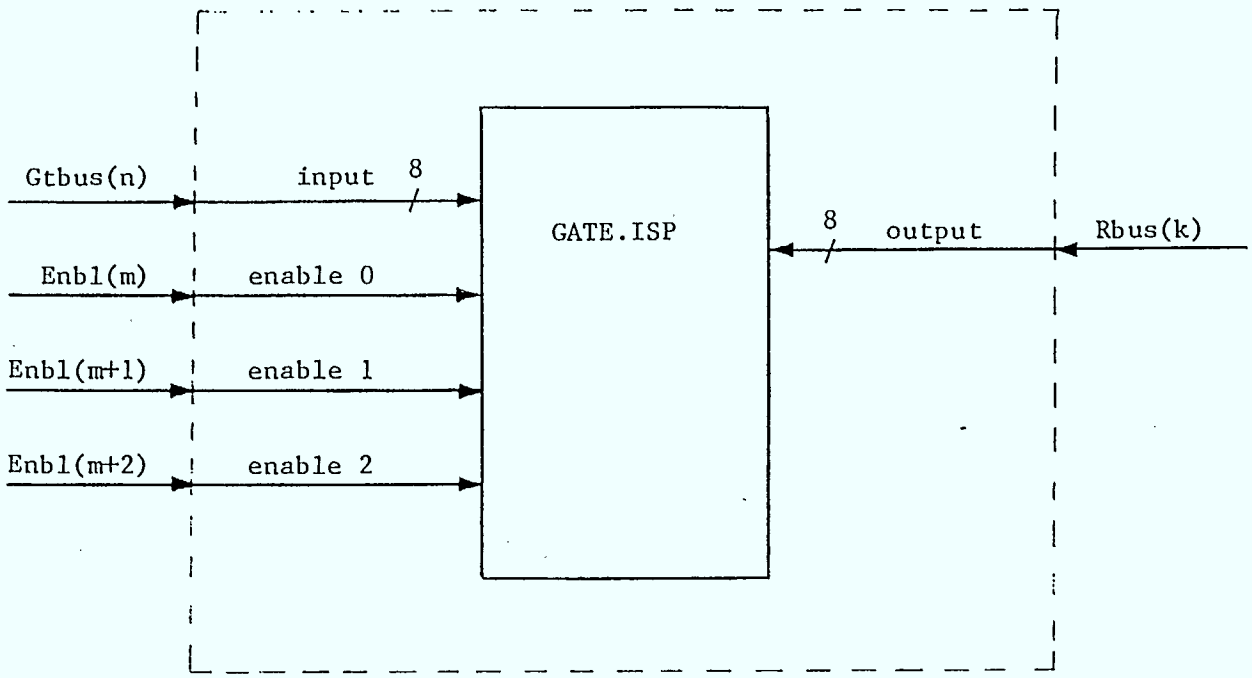
The two types of guardians as described above have been implemented and tested in N.mPc. A detailed description of ports and interconnection within the system of gate and guardian modules is given by Figures 2-25 and 2-26. Note that a guardian attached to a peripheral always starts in the OFF state whereas a guardian attached to an interface processor may start either in the OFF or in the ON state. In actual hardware this will be controlled by a jumper or a switch. In the simulation some guardians always start in the ON state. Namely the guardians that start in the ON state are selected so that each of the three initially active interface processors will have access to one of the initially active buses. This is essential for the system to be operational after initialization. It is now necessary to discuss the guardian initialization process in more detail. The initialization process provides all guardians with an identification number and informs every interface processor guardian whether it should initially be turned on or off. In actual hardware one would simply use a dipswitch per guardian in order to distribute an ID and initialization information to each guardian. As there would be independent switches for each guardian no single point of failure will be introduced. However, in the



PPG: Peripheral Processor Guardian

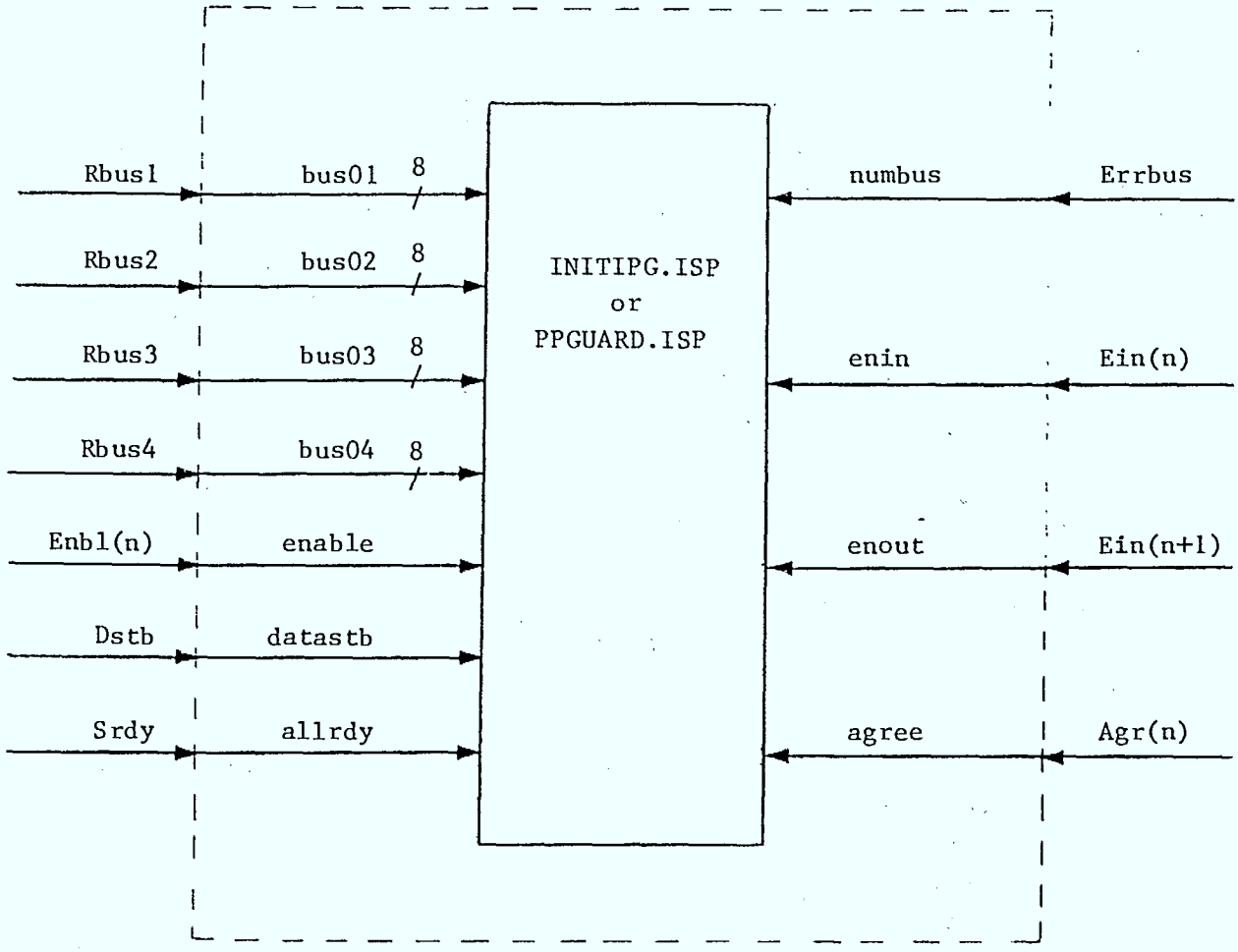
IPG: Interface Processor Guardian

Figure 2-24: Example of Guardian Operation



NB: (k), (n) and (m) are numbers which depend on what other modules are connected to a gate.

Figure 2-25: The Gate Module



NB: The number (n) indentifies a guardian.

Figure 2-26: The Guardian Module

simulation context the dipswitches were replaced by an initialization mechanism that was easier to build and handle than sixty dipswitch modules. Figure 2-27 shows the initialization circuitry of the guardians. An initialization module distributes the initialization information using a special bus ("numbus"). Through another line ("enin", "enout") a token is passed from guardian to guardian as each one receives its initialization word. A "reset" signal allows to reinitiate the guardian initialization whenever necessary. The most significant bit of the initialization word is recognized by the interface processor guardians (only) and tells them whether they should initially be switched on or off. The rest of the initialization word is used to distribute ascending identification numbers to each guardian. Multiples of four are not used as guardian identifiers. This measure makes it possible to address all three guardians (plus the nonexisting one with an ID that is a multiple of four) associated with a certain gate with one single command. In the guardians this feature is implemented by having them recognize commands based on an ID whose two least significant bits are truncated. Figure 2-28 shows the initialization module in detail. It contains the IP-guardian initialization information and may therefore be different if different hardware is simulated. The initialization also distributes identifications to all transmitters and receivers via ports that are initialized to a certain value using the ISP' initialization mechanism. The transmitter/receiver modules get these values by connecting a line to the appropriate initialization module port.

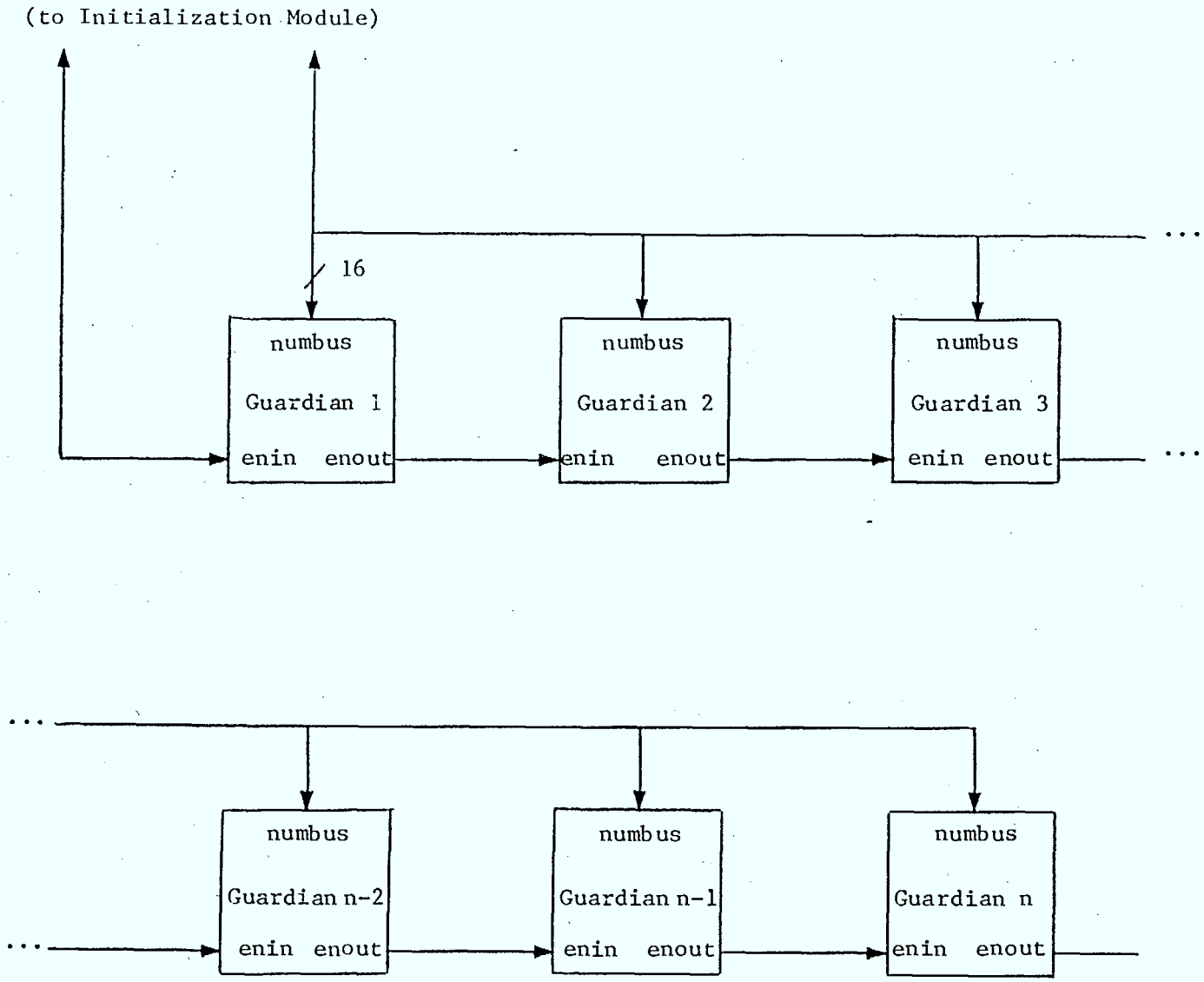
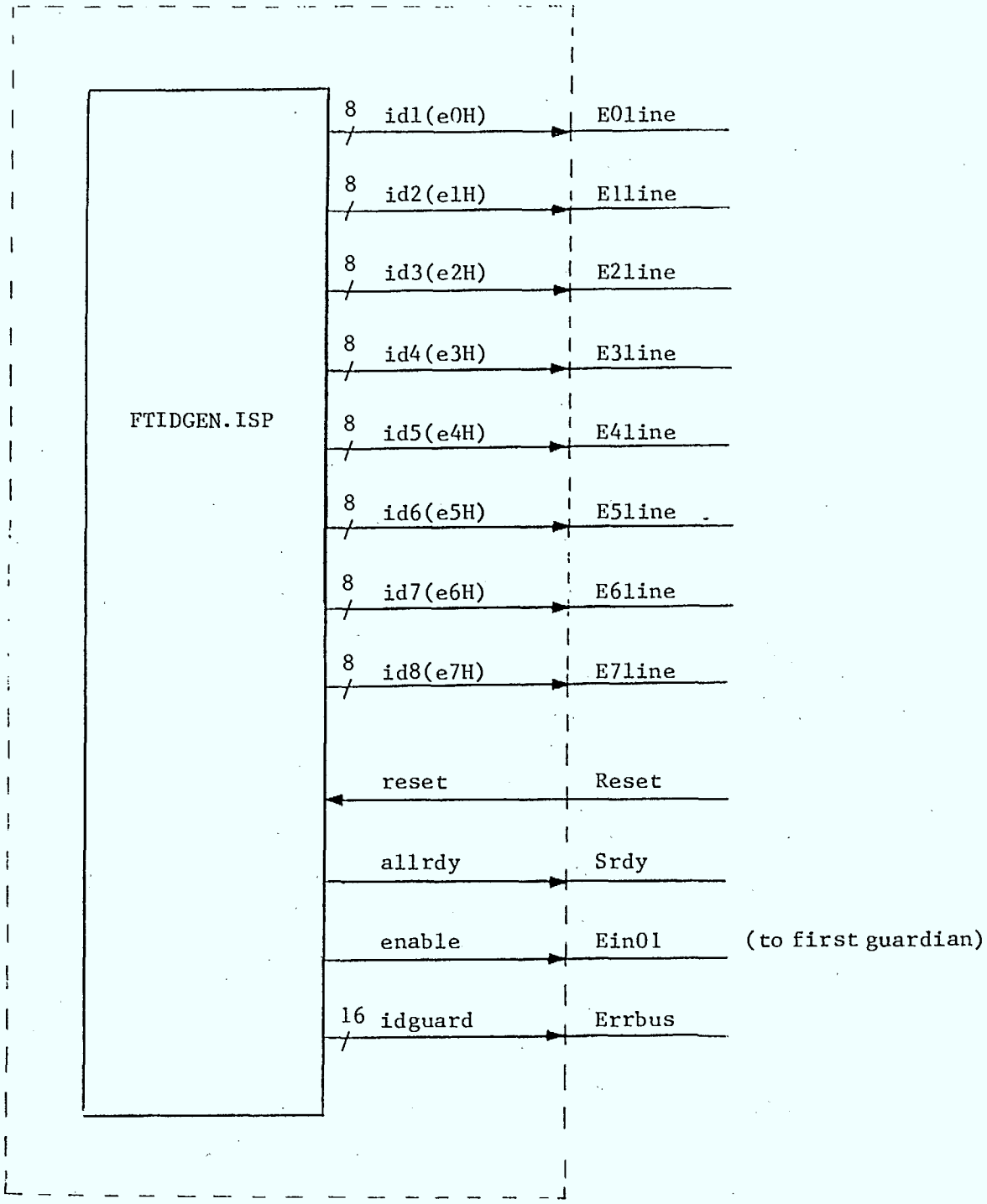


Figure 2-27: Initialization of the Guardians



NB: The numbers in brackets are the initial values assigned to a port by an N.mPc mechanism. This corresponds to a "hardwired" initialization

Figure 2-28: The Initialization Module

2.3.3. The System Reset Mechanism

The system reset mechanism description given in [14] was formulated with a view towards system implementation in real hardware. Thus "powering down" of microprocessors and running of self test routines were not considered in the context of the simulation work presented in this report. However, the hardware simulation has the capability of reactivating the guardian initialization mechanism. This capability is described in Section 2.3.2.3.

2.4. Fault Tolerant Operating System Implementation

The fault tolerant operating system has been developed keeping in mind that it should be easy to port it from its host computer based development environment (described in [7]) to some specific target hardware. Therefore the only hardware dependent communication routines are four utility routines called by the message passing kernel for doing inter processor communications.

The communications routines necessary to run the fault tolerant operating system on Intel 8086 based hardware (simulated or real) were written and tested on simulated hardware and are described in section 4.

2.5. System Integration

This section explains how to integrate the fault tolerant operating system with the simulation of the fault tolerant hardware architecture in order to form a fault tolerant multi-microprocessor system.

The first step is to design descriptions of all the necessary hardware components in N.mPc's hardware description language ISP'. These components are then interconnected according to the design of the fault tolerant multiprocessor architecture and the interconnection

information is stored in a topology file. Figure 2-29 shows the hardware modules involved in the simulation of the fault tolerant multiprocessor architecture as well as their interconnection.

The next step is to load the operating system software in the processor's memories. The enhanced software development environment (C-8086 cross compiler) described in [20] is used to load the software of the fault tolerant operating system, written in the C language, into the memories of all central processors. For the interface processors and the peripheral processor special drivers written in assembly code are loaded. Software and hardware of the entire system can now be integrated under N.mPc as shown in Figure 2-30. The simulation is then put in the runtime mode and the processors are initialized appropriately. The simulated fault tolerant multiprocessor system will be ready to run at this point.

The methodology outlined above for the integration of the operating system code with the simulated hardware architecture is general and will definitely lead to a fully integrated system simulation. In this study the full integration was not accomplished due to the following reasons:

- The slow execution speed of N.mPc mitigates against conducting fault tolerant tests in a reasonable time period. This is due to the fact that the execution of each single operating system instruction involves detailed and numerous register transfer level instruction executions with N.mPc. Since the task execution cycles of the operating system are fairly lengthy to begin with, the corresponding number of executions within N.mPc will be extremely large, which leads to prohibitively long simulation times.

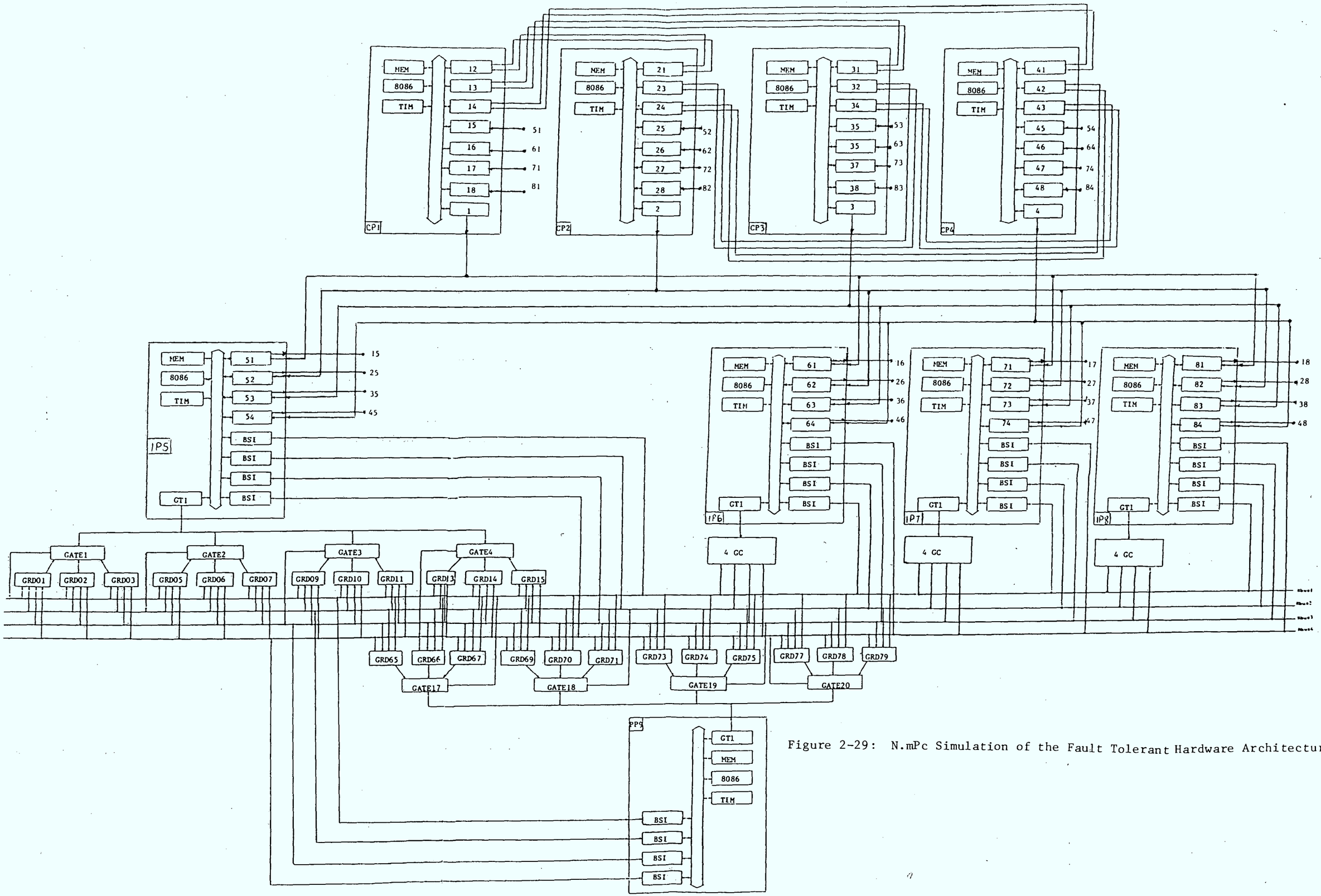


Figure 2-29: N.mPc Simulation of the Fault Tolerant Hardware Architecture

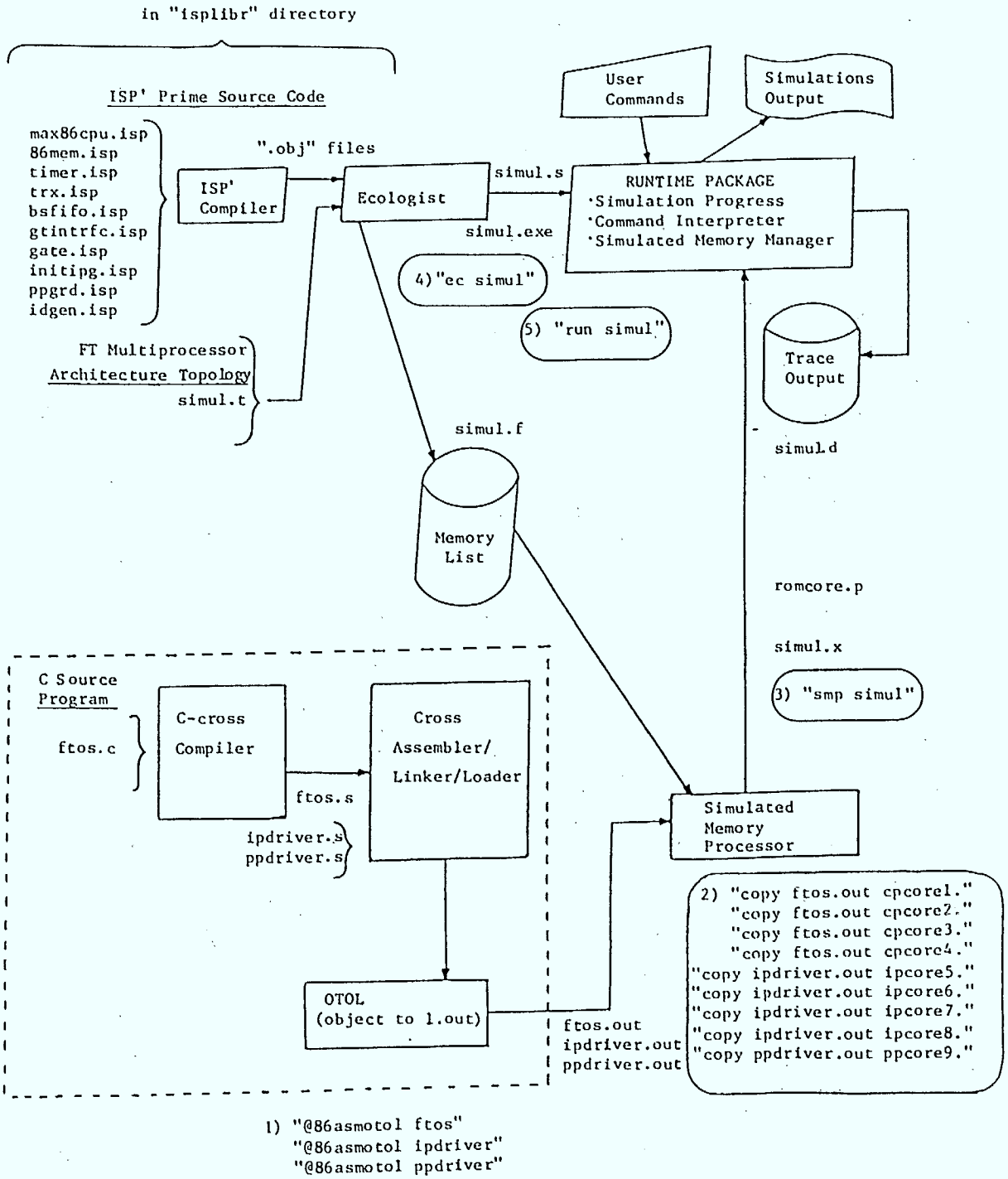


Figure 2-30: Integration of the Fault Tolerant System

- The full operating system code, written in C programming language, proved to be lengthy and complex which makes the debugging process of the code, when executed under N.mPc, intractable.
- The N.mPc code corresponding to the VLSI structure of the 8086 chip details is not guaranteed to be absolutely correct. This complicates the debugging process and makes it difficult to determine whether the source of a given bug is in the VLSI description or in the complex operating system code. Note that this problem is inherent to any complex VLSI design such as the 8086 processor.

In summary a full integration of the fault tolerant hardware and the operating system software proved to be beyond the time and resources allocated for this study.

The functionality of the operating system software was tested separately in a "C" based testbed as described in [7]. In the simulation of the fault tolerant multiprocessor system the full operating system software was replaced by a module exercising the hardware dependent communication routines used by the message passing kernel. The testing of the fault tolerant hardware and the application of different failure conditions to the simulated system have been conducted under the N.mPc environment in order to validate the concepts developed in [7, 14].

3. FUNCTIONAL TESTING OF THE SIMULATED HARDWARE SYSTEM

3.1. Simulation Limitations

In N.mPc based hardware simulations the execution speed is inversely proportional to the complexity of the simulated hardware. In microprocessor based simulations this means that the execution speed of the total simulation will decrease in proportion to the number of microprocessors simulated. Validation of the architectural hardware concepts does not necessitate simulation of the entire fault tolerant hardware architecture as many modules are repeated in the structure. For this reason, the simulation has been broken down into smaller modules. This will speed up each simulation run without affecting the generality of the obtained results. The following module simulations have been created and are shown in Figure 3-1:

i) A simulation involving an interface processor, a peripheral processor and their gate complexes.

ii) A simulation with four interface processors, one peripheral processor and all the fault tolerant hardware.

iii) A simulation of four fully interconnected central processors and their communication modules.

Simulation i) is sufficient to exercise the basic properties of the elements of the fault tolerant hardware. Simulation ii) can demonstrate the synchronous functioning of the interface processors and makes it possible to test the fault tolerant hardware architecture in different failure conditions. Simulation iii) is used for the integration of the hardware dependent communication routines of the fault tolerant operating system with the simulated hardware architecture.

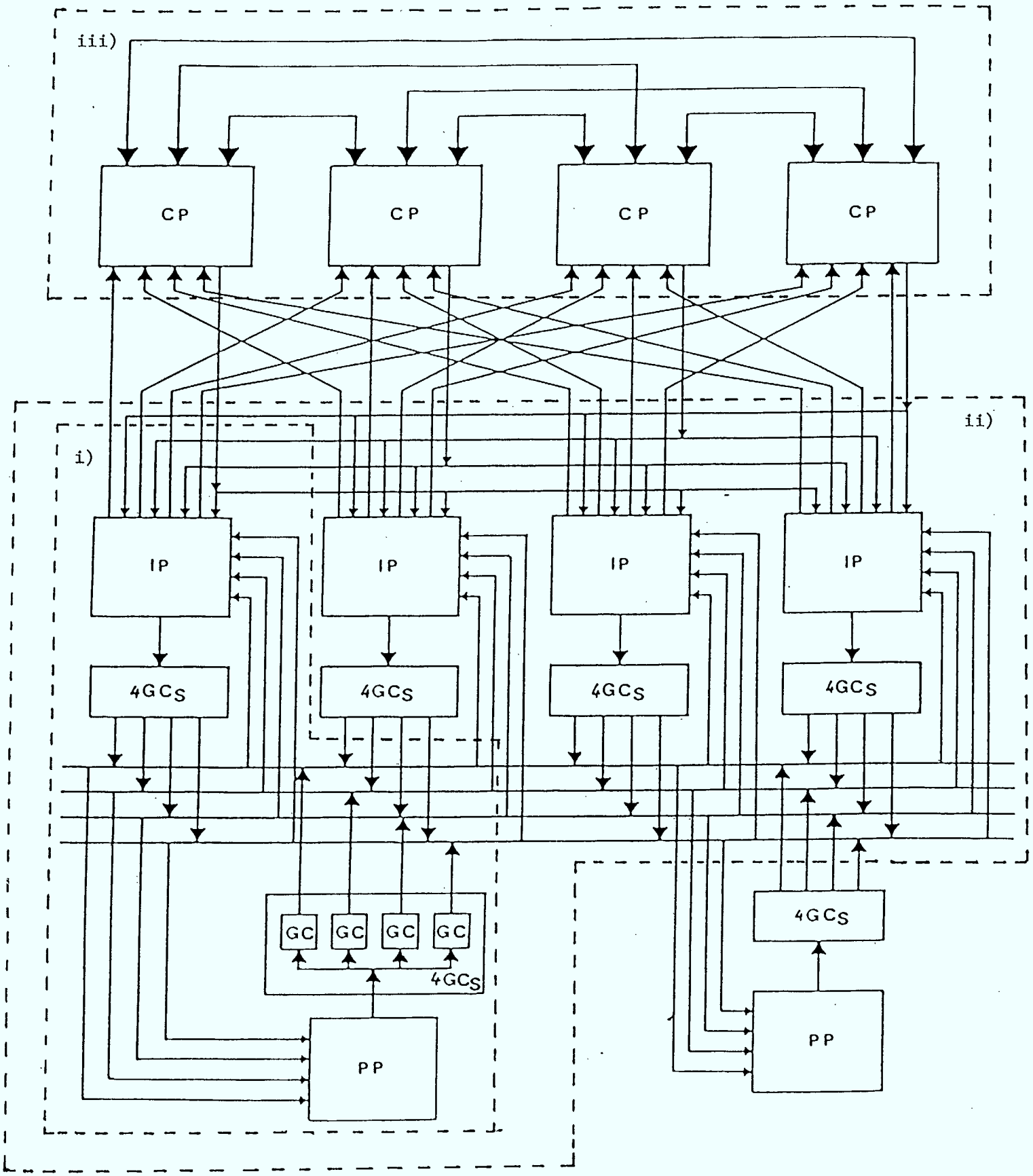


Figure 3-1: Partial Simulations of the Fault Tolerant Multiprocessor Architecture

3.2. Test Plan and Approach

3.2.1. Objectives

The main objective of the simulation is to establish the correctness and completeness of the detailed descriptions of the hardware modules. Correctness of the interconnection (topology) of these hardware modules is also established through appropriate simulation tests.

A second objective of the simulation is to demonstrate the ability of the fault tolerant hardware to continue correct operation in the presence of (induced) faults and to perform a system reconfiguration upon detection of a hardware failure.

3.2.2. Test Plan

The test procedure includes two steps. First the hardware modules and the guardian initialization mechanism are tested in a simulation involving one interface processor and a peripheral processor as well as all the fault tolerant hardware. This test should assess the functional correctness of the hardware modules. The test method applied consists in having appropriate test software exercise all the features of each hardware module. The result expected from this test should give a confirmation that the fault tolerant hardware needed for the multiprocessor architecture works exactly as specified in [14].

In a second test simulation, involving all four interface processors, the synchronous nature of the interface processors will be demonstrated. The fault tolerant system's ability to continue functioning correctly in the event of corrupted data, guardian malfunction or bus failure will be demonstrated. This test should also demonstrate the correct functioning of the full fault tolerant hardware

modules used to interface the central processors and the peripherals. The test method used in this simulation is selective fault insertion by the user in order to test the fault tolerant multiprocessor system's response. The expected results were the masking of single errors in input data by majority software voting, the cutting off of a device from its bus when one of its guardians fails and the reconfiguration of the fault tolerant hardware around the failure of a peripheral interface bus.

3.3. Test Software

3.3.1. Functional Definition

3.3.1.1. Test of the Hardware Modules

Testing the hardware modules is conducted via a simulation which involves an interface processor and a peripheral processor as shown in Figure 3-2 inside the enclosed area (i). Test software modules are needed to drive each processor. The listings of the interface processor driver ("iptest.s") and the peripheral processor driver ("pptest.s") used in this test simulation are found in Appendix A.2.

The "iptest.s" program exercises all the states of the guardians shown in Figures 2-20, 2-22. It tests all guardian commands shown in Figures 2-21, 2-23. In cooperating with a program driving the peripheral processor the whole guardian operation sequence shown in Figure 2-24 was exercised in the test. The "pptest.s" module just waits until the peripheral processor guardians are enabled and then sends a message to the interface processor via the temporarily accessible peripheral interface bus. The guardian initialization mechanism, which is independent of the test software, is also checked by this test simulation.

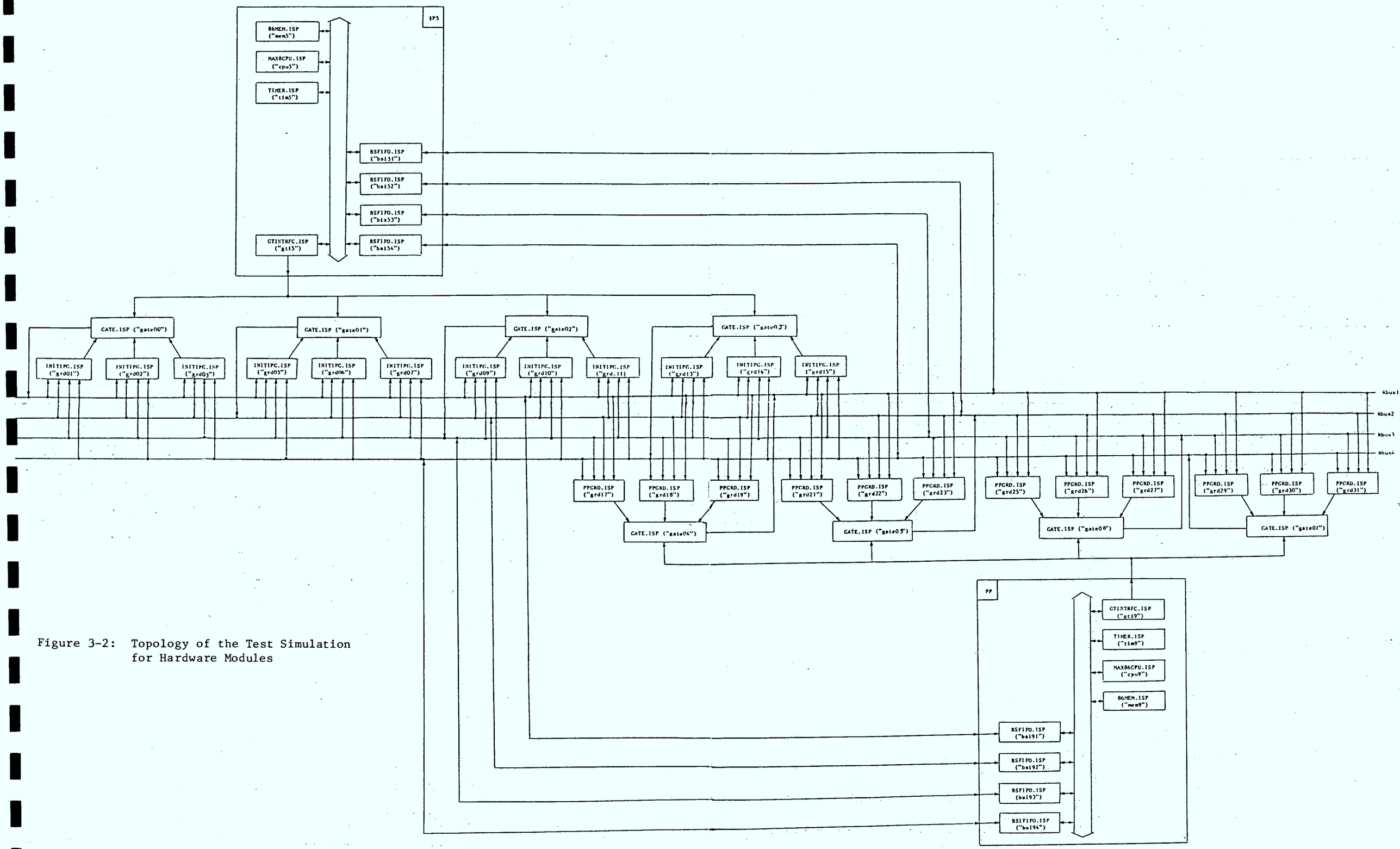


Figure 3-2: Topology of the Test Simulation for Hardware Modules

3.3.1.2. Test of the Full Fault Tolerant Interface

This simulation is intended to demonstrate the fault tolerant system's capability to continue correct operation in the presence of specific user induced faults. The full fault tolerant interface, including four interface processors and a peripheral processor, is part of the simulation. The configuration is shown in Figure 3-1 inside the enclosed area (ii).

The program driving the four interface processors ("fttest.s") does a majority vote on input data and then goes through normal operation of the fault tolerant interface and its guardians as shown in figure 2-24. The peripheral processor waits until it gets access to a peripheral interface bus, which means it has to wait until its guardians are enabled. It then sends data to the interface processors. If the user does not insert any faults this process continues indefinitely. The user can then insert simulated faults, causing the fault tolerant hardware and software to react in order to keep the system operating correctly. Three failure conditions have been simulated:

1) The user can corrupt input data. A subsequent majority software vote in the interface processors can mask a single error.

2) The user can change the gate enable output signal of a guardian to an incorrect state. Such a fault is detected and masked by the guardian's gate, which only opens if its three guardians are in agreement.

3) The user can cause a disagreement among the three peripheral interface buses in use. This will be interpreted as a bus failure by the interface processors and the fault tolerant

system is reconfigured around the failure by bringing in the fourth reserve bus.

The results expected from this test are a continuing correct operation of the fault tolerant system in presence of the above mentioned user induced faults and a correct system reconfiguration upon detection of a bus failure.

3.3.2. Implementation Description

Detailed descriptions of how to cross compile and run the test software modules discussed in section 3.3.1 are given in Appendix A.3 as well as in the simulation directories ("persim", "ftol") in the form of "readmefirst" textfiles. The directories reiterate what has been explained in the N.mPc user's manual[24] for the specific cases of the two simulations discussed in this section.

3.4. Test Results

3.4.1. The Hardware Module's Test Simulation

The results of the simulation discussed in this section are recordings of certain events (register transfers, signal changes, etc.) which simulate the function of a complex hardware.

Figure 3-2 shows the topology of the test simulation for the hardware modules and Figure 3-3 gives the usual block diagram of the same simulation.

The listing of an interactive test simulation session , focussing on the verification of the correct functioning of the fault tolerant hardware modules, is given in Appendix A.4. It contains all the information necessary to verify that the modules under investigation do in fact work as specified in [14]. N.mPc makes it then possible to verify with "display" commands and breakpoints that the hardware is

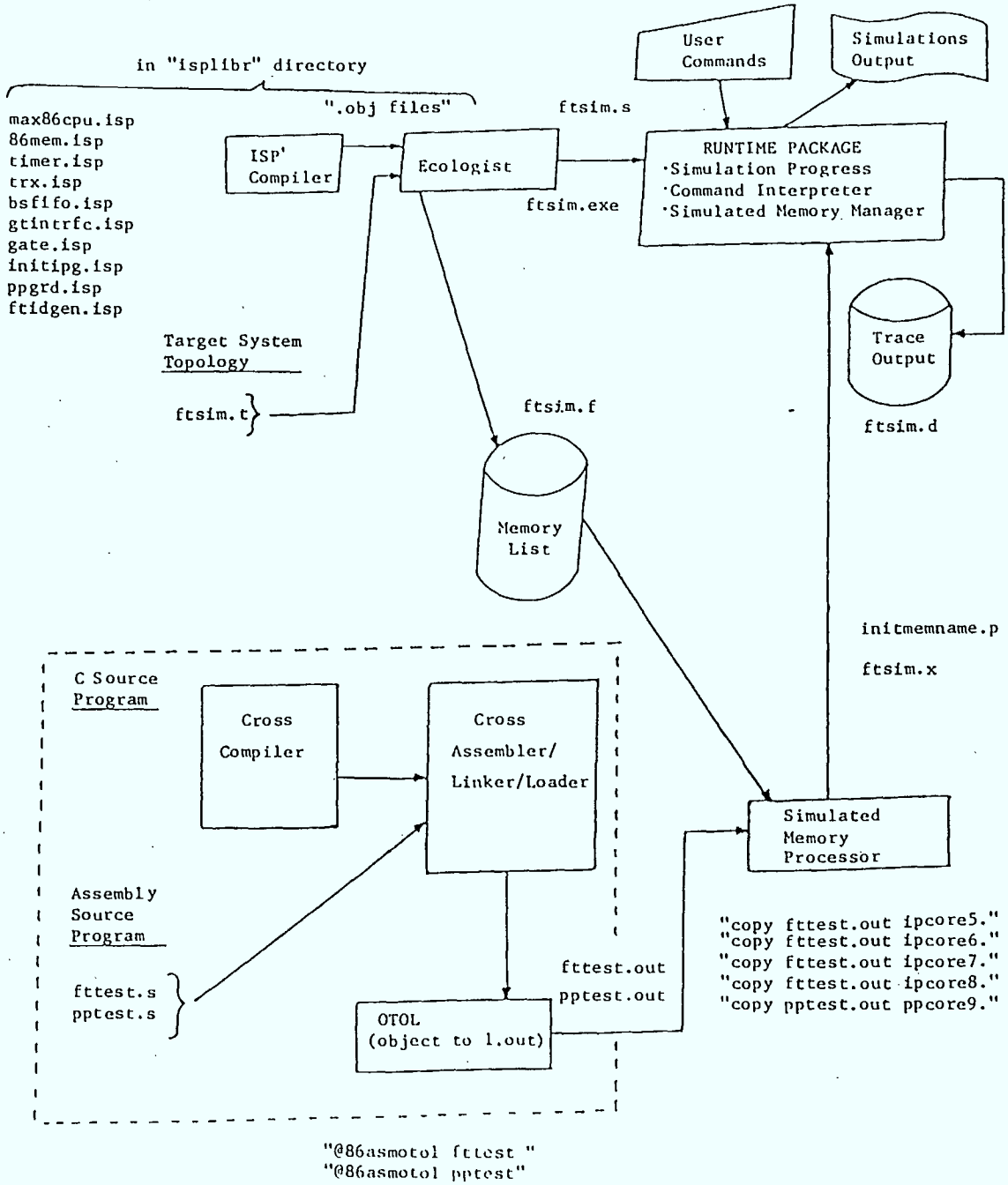


Figure 3-5: Block Diagram of the Test Simulation for the Full Fault Tolerant Interface

functioning correctly. The following conclusions could be drawn from simulation test runs:

- interface processor guardian ("initipg.isp"):
  - all state transitions done according to state diagram
  - it executes all its special commands correctly
  - initial ID distribution and initialization (on/off) working correctly
  - it activates its enable signal only when turned on during the initialization or after receiving the "Turn On" command
- peripheral processor guardian ("ppgrd.isp"):
  - all state transitions done according to state diagram
  - it executes all its special commands correctly
  - initial ID distribution functions
  - activates its enable signal only temporarily after receiving a "Select" command followed by an "Enable" command
- gate("gate.isp"):
  - opens only when enabled by all three guardians
- initialization module ("idgen.isp"):
  - correctly turns the desired interface processor guardians on

- distributes IDs to all guardians
- distributes the right I/O addresses to each communication module

- transmitter("gtintrfc.isp"):
  - correctly transmits the bytes written to its I/O address via a FIFO queue to peripheral interface bus(PIB, redundant bus)
  - the CPU can check whether the transmitter queue is full before enqueuing another byte
- receiver("bsfifo.isp"):
  - correctly puts bytes received from the PIB in a receiver queue
  - the CPU can check whether the receiver queue is empty before reading a byte from it
- transceiver("trx.isp"):
  - correctly transmits and receives bytes using two separate FIFO queues
  - a CPU can check whether the receiver queue is empty before reading from it
  - a CPU can check whether a transmitter queue is full before writing to it
  - a CPU can reset both FIFO queues

- timer("timer.isp"):
    - the timer correctly halts a CPU for a number of clock cycles that can be specified
- memory("86mem.isp"):
    - the 16k RAM memory has been tested in the course of earlier N.mPc work(see [20])
- 8086 CPU("max86cpu.isp"):
    - the description of an Intel 8086 CPU has been the object of earlier N.mPc test and verification work(see [20])

3.4.2. Results of the Test of the Full Fault Tolerant Interface

The topology of the simulation discussed in this section is depicted in Figure 3-4 and the corresponding simulation block diagram is presented in Figure 3-5. Again the "results" are not representable in a simple, closed form. They represent changing states in a complex simulated hardware that can only be observed by using various N.mPc run-time commands in order to trace certain registers or ports.

The listing from an interactive N.mPc simulation session is contained in Appendix A.4. It shows the testing of three different failure conditions that can be handled by the fault tolerant hardware of the simulated multiprocessor architecture. The following conclusions can be drawn regarding the three fault conditions specified in 3.2.2:

1) Corrupted Input Data:
    - Data fed to interface processors (simulating a message from three central processors) can selectively be

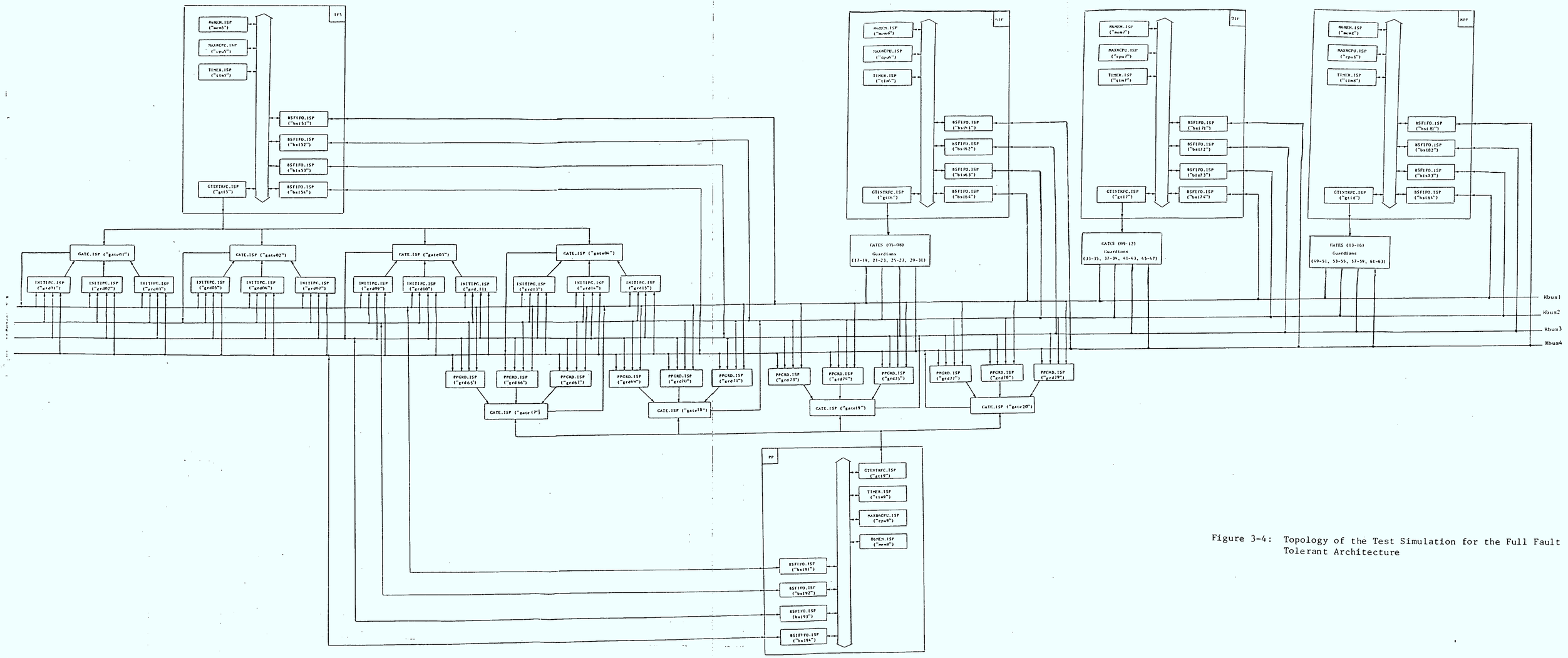


Figure 3-4: Topology of the Test Simulation for the Full Fault Tolerant Architecture

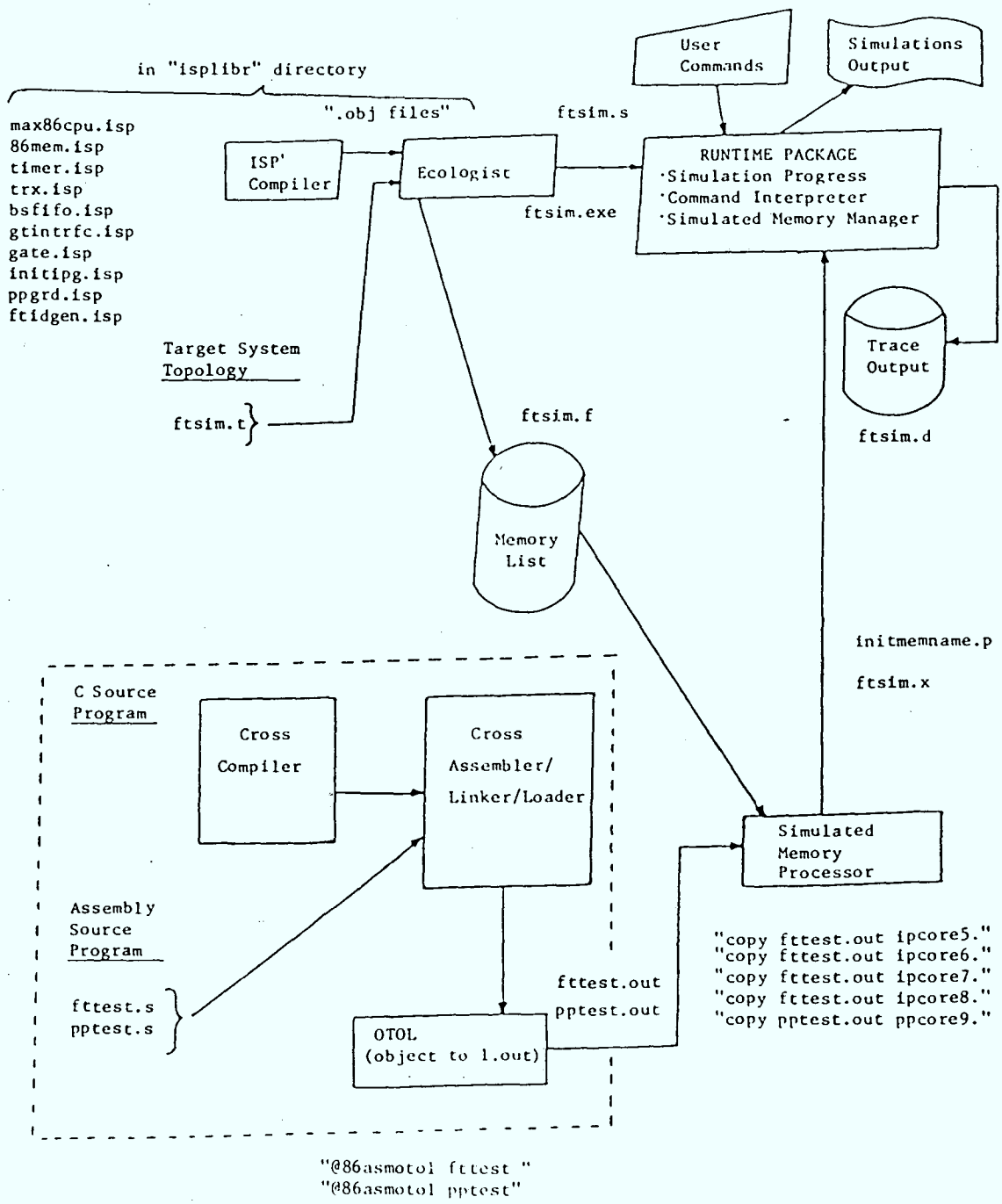


Figure 3-5: Block Diagram of the Test Simulation for the Full Fault Tolerant Interface

corrupted. The interface processor can mask a single error by majority software voting.

2) Guardian Failure:

- The failure of a guardian can be induced by the "user"(= person running the simulation), who inverts the logical state of a guardian's enable signal. This failure is correctly handled by the fault tolerant multiprocessor system as a gate gives a device bus access only if enabled by all of its three guardians. Thus the device controlled by the faulty guardian is cut off from its bus.

3) Bus Failure:

- A disagreement between the three redundant buses in use at any time is detected by the interface processors. They can then reconfigure the system by determining the faulty bus and switching the appropriate guardians and the concerned interface processor to a previously unused reserve bus. Figure 3-6 and 3-7 show the configuration of the fault tolerant hardware before and after recovery from a bus failure.

More details about these simulations are found in "0readme.fst" information files contained in each simulation directory, in comments in the listings of the programs run in each simulation and in the listings in Appendix A.4.

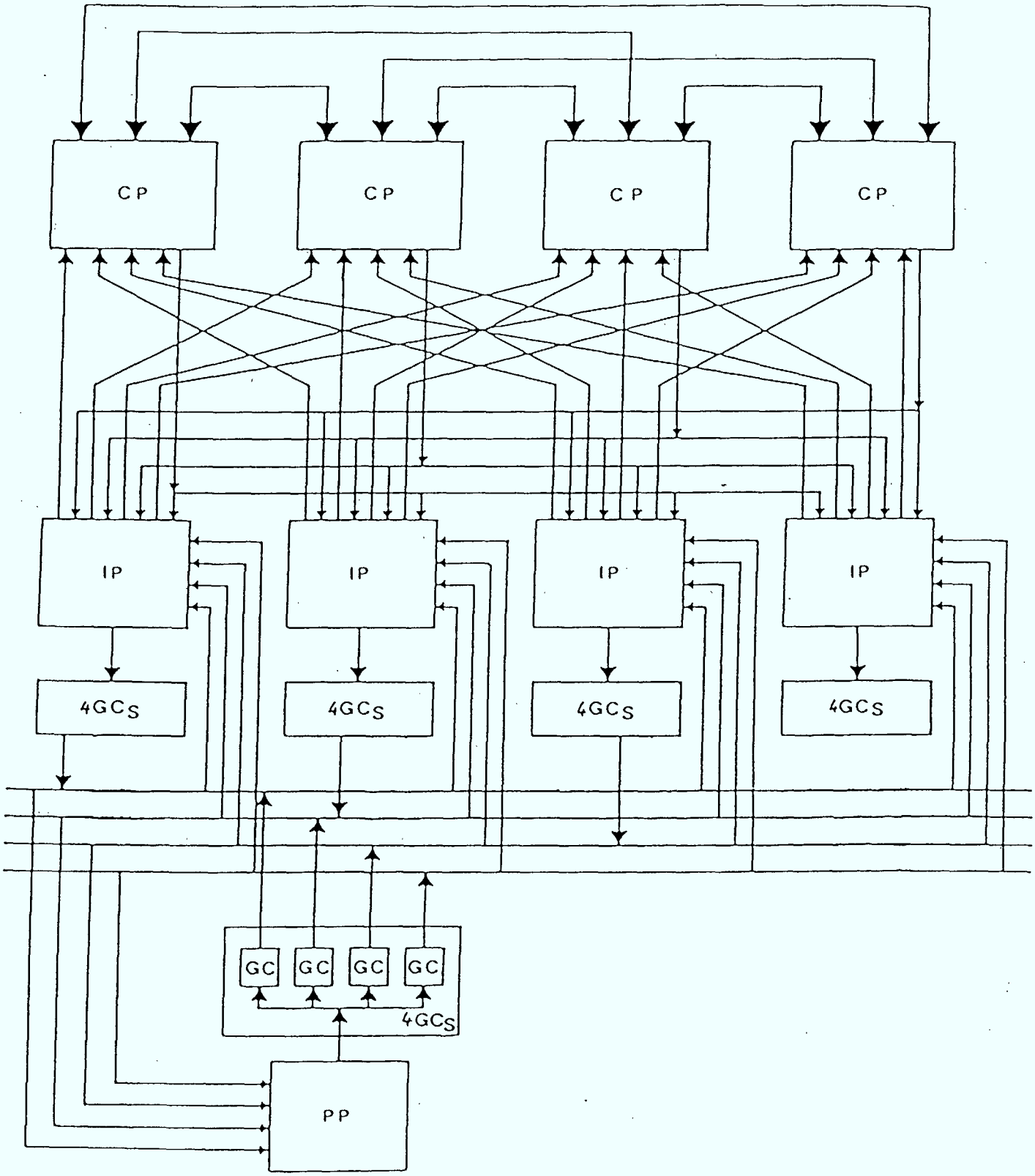


Figure 3-6: Initial Configuration of Fault Tolerant Hardware

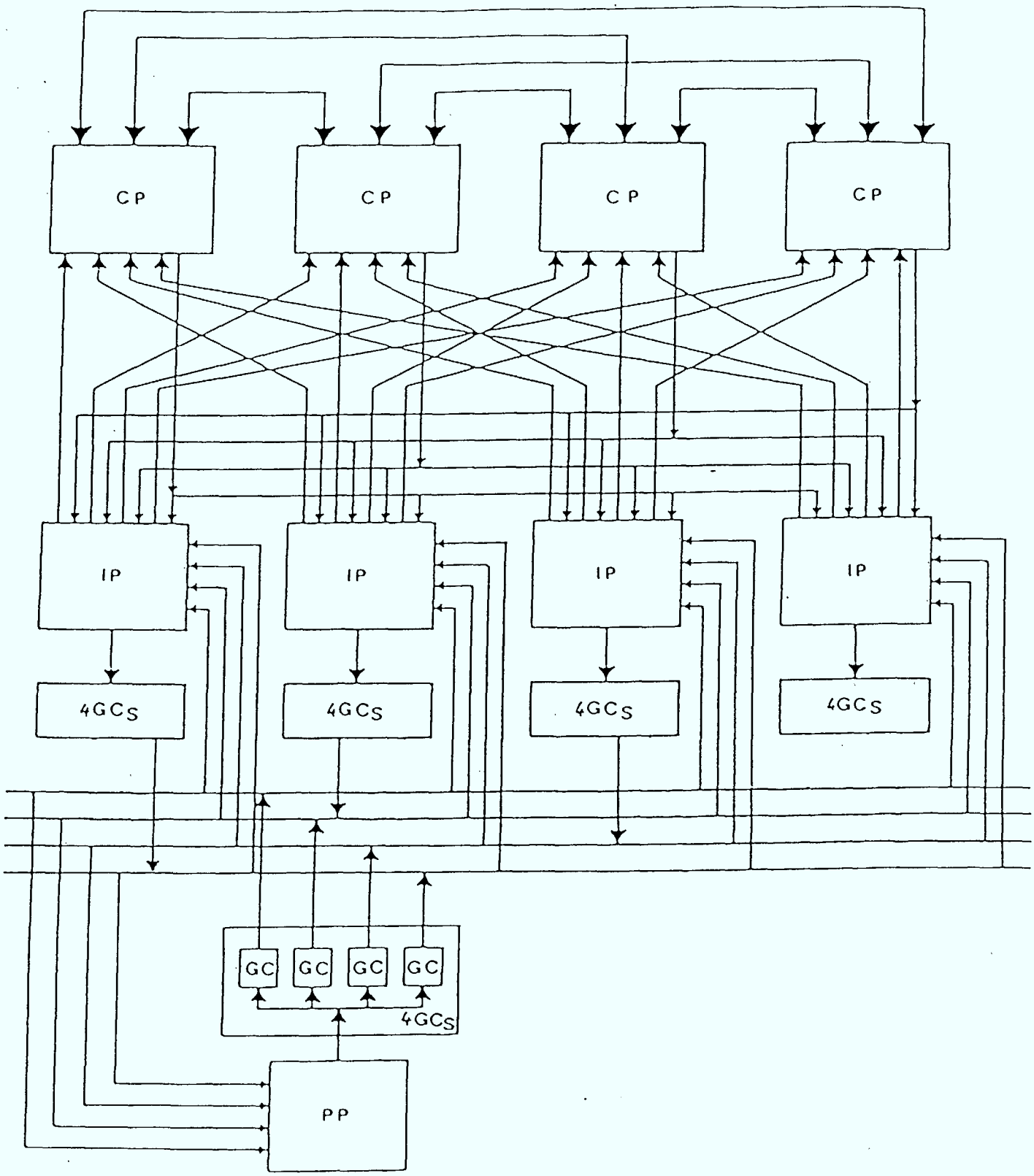


Figure 3-7: Reconfigured Fault Tolerant Hardware after Detecting a Failure of Peripheral Interface Bus Number 1

4. FUNCTIONAL TESTING OF THE INTEGRATED OPERATING SYSTEM SOFTWARE AND THE SIMULATED HARDWARE SYSTEM

4.1. Limitations of N.mPc Simulations

The originally planned integration of the simulated multiprocessor architecture and the full fault tolerant operating system was subsequently scoped down to a simulation involving the integration of the fully simulated multiprocessor hardware architecture and only a subset of the operating system routines for the following reasons:

i) The work done on the validation of N.mPc microprocessor simulation [20] concluded that a substantial host processor CPU time is required when executing software on a simulated processor instead of a real processor. Since the fault tolerant operating system was still in the conceptual design stage at the time the validation work was being conducted, it was difficult to estimate the performance of N.mPc when executing a complex software structure such as the fault tolerant operating system. The simulation described in this section gives accurate estimates of the magnitude of the CPU resources needed to simulate software modules that are run on simulated microprocessors. This simulation includes four central processors communicating with each other by exchanging messages. While the simulation works as described in the hardware specifications stated in [14] it also indicates clearly that the host CPU execution time needed for running the complete fault tolerant operating system software on simulated 8086 processors would be very high. For example, the exchange of a single byte "message" consumes close to one minute of CPU time on the VAX 11/780 host computer.

Considering that the lowest layer of the fault tolerant operating system involves frequent exchange of messages each with length in the order of hundreds of bytes, it is easy to estimate that the CPU time required to perform a meaningful simulation will be unrealistically large.

ii) The slow execution speed of N.mPc mitigates against conducting fault tolerant tests in a reasonable time period. This is due to the fact that the execution of each single operating system instruction involves detailed and numerous register transfer level instruction executions with N.mPc. Since the task execution cycles of the operating system are fairly lengthy to begin with, the corresponding number of executions within N.mPc will be extremely large, which leads to prohibitively long simulation times.

iii) The N.mPc code corresponding to the VLSI structure of the 8086 chip details is not guaranteed to be absolutely correct. This complicates the debugging process and makes it difficult to determine whether the source of a given bug is in the VLSI description or in the complex operating system code. Note that this problem is inherent in any complex VLSI design such as the 8086 processor.

In summary a full integration of the fault tolerant hardware and the operating system software proved to be beyond the time and resources allocated for this study.

For the reasons described above, it was decided to implement only a subset of the fault tolerant operating system. The subset consists of hardware dependent communication routines running on a simulated hardware architecture which consists of four intercommunicating central

processors. This partial simulation is based on the fact that the fault tolerant part of the multiprocessor hardware architecture has been tested before. The breakdown of the total simulation into partial simulations is necessary to complete the simulation while imposing a reasonable demand on the host computer CPU time. The hardware dependent communication routines of the fault tolerant operating system were chosen for integration with the simulated hardware for two reasons:

i) The hardware dependent communication routines could be easily transported in the future as part of the full operating system software to an 8086 based prototype hardware.

ii) It is difficult to simulate separately any of the other operating system routines since they are all interrelated.

4.2. Test Plan and Approach

4.2.1. Objectives

The main objectives of the simulation described in this section are:

(1) To verify the correctness and establish the completeness of the operating system routines used for interprocess message communications. This will also establish that the interface between the hardware and the software responsible for byte transfer among the different processors is correct.

(2) To establish systematic and methodical procedures for transporting operating system code, written in C, to a target hardware prototype. This will be desirable in future research involving the development of actual fault tolerant multimicroprocessor hardware based on the study completed so far.

4.2.2. Test Plan

The following activities were planned in the context of the integration of the operating system and the fault tolerant hardware architecture:

- Investigation of the mechanism for calling hardware dependent assembly routines from within hardware independent high level software.
- Design and test of special hardware modules ("transceivers") for two way message passing between the fully interconnected central processors following the criteria defined in the hardware analysis report [14].
- Write the assembly routines necessary for message passing between central processor modules, using the previously designed transceivers.
- Write moderately sized high level test software modules (in C) coordinating message passing between four central processors.

The expected result of this test consists in the integration of all the message passing software and the four central processor modules in order to build an interprocessor message passing simulation that can be executed on the N.mPc/VAX 11-780 (located at CRC).

4.3. Test Software

4.3.1. Functional Definition

Figure 4-1 gives an overview of the different software and hardware modules involved in the simulation of four fully interconnected, communicating central processor modules.

The test software running each of the four central processors is written in the high level language C and organizes the message exchange

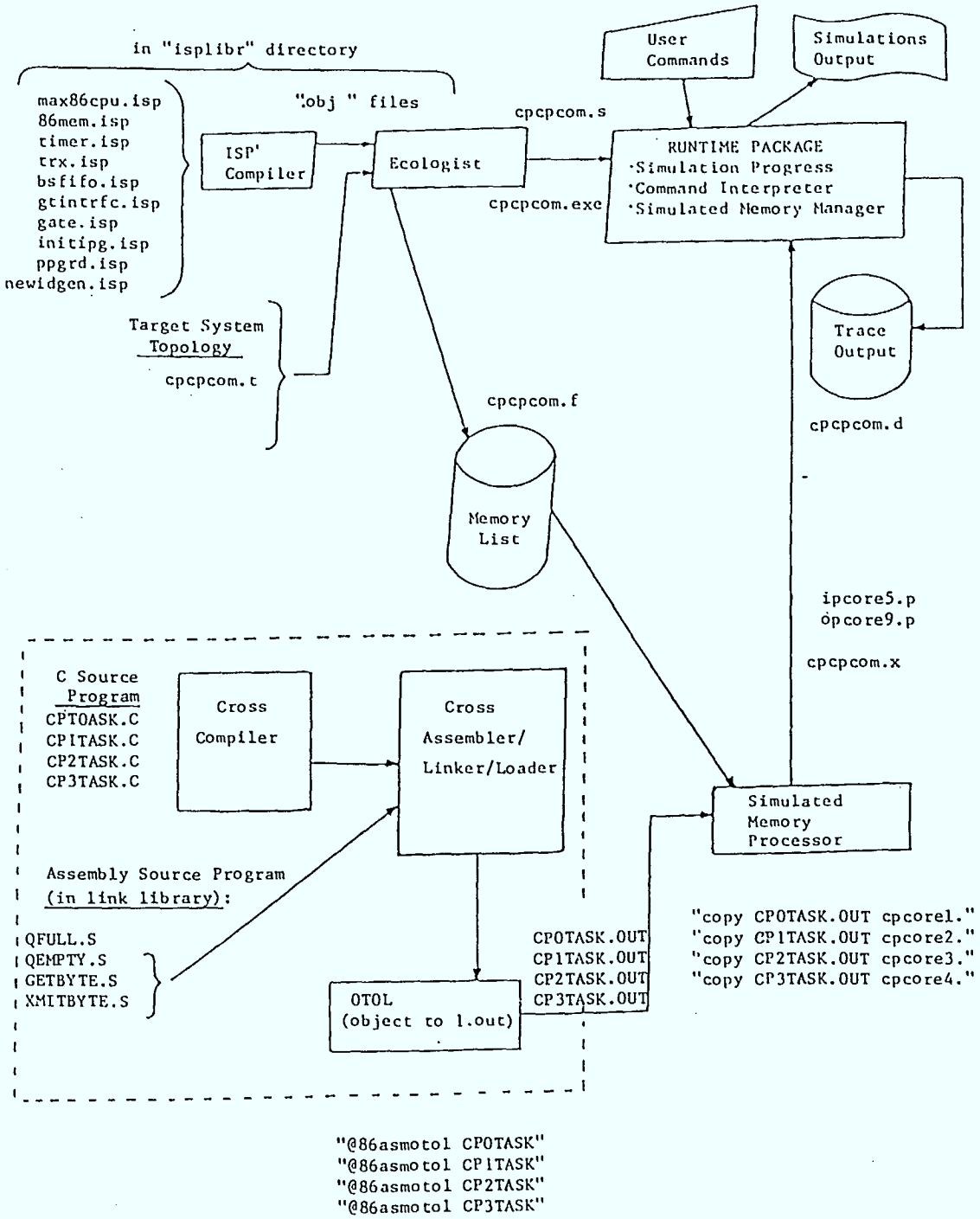


Figure 4-1: Block Diagram of the Fault Tolerant Operating System/Hardware Integration Simulation

between the four processors. Appendix B.3 contains the listing of each processor's message passing software module. A message of six bytes in length is passed around from processor to processor in a closed loop as shown in Figure 4-2.

The hardware independent high level software modules running on the central processors naturally can't do the message passing by themselves. They need to call hardware dependent assembly routines that are able to handle the communications hardware modules ("transceivers", trx.isp) described in section 2. The four assembly routines for communication are:

- "Qfull.s": Checks whether the transmit queue of a transceiver is full or not. A "1" is returned if the transmit queue is full, a "0" if the queue is not full.
- "xmitbyte.s": Puts a specified byte into the transmit queue connected to the desired destination processor. The byte is then automatically transmitted to the receive queue of the destination processor.
- "Qempty.s: Checks whether the receive queue of a transceiver is empty or not. A "1" is returned if the receive queue is empty, a zero otherwise.
- "getbyte.s": Reads a byte from the receiver queue connected to the desired source processor.

The four communication routines have to be incorporated into the link library of the cross software development tool (see [19, 20]). They can then be called from within programs written in the high level

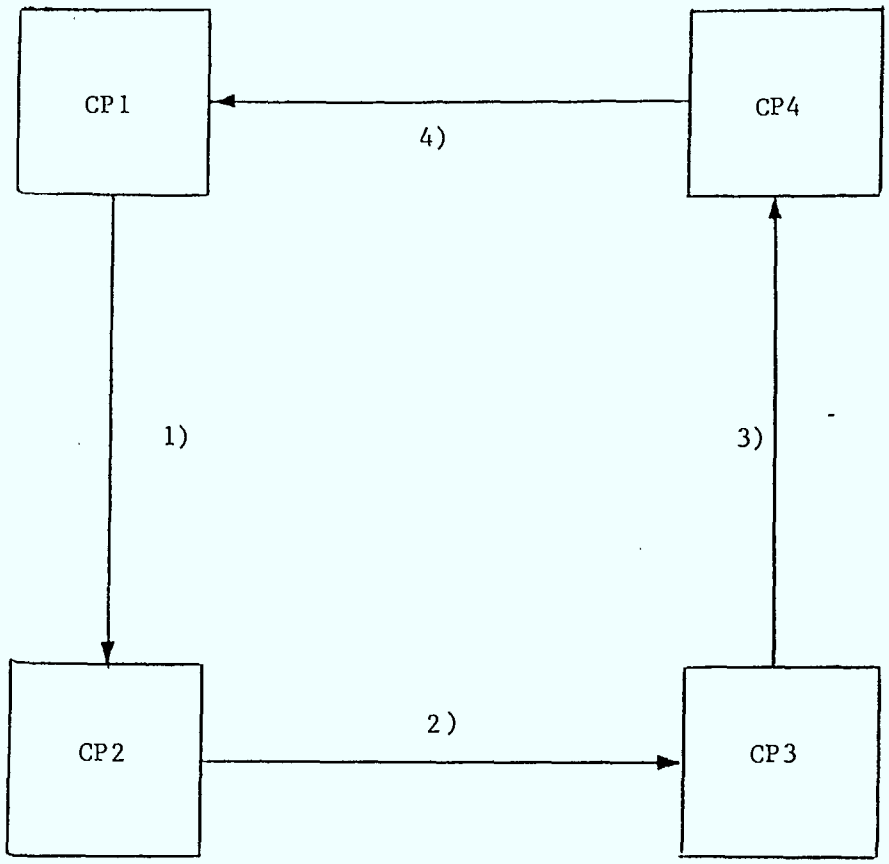


Figure 4-2: Message Passing Loop Formed by Four Central Processors

language C. Assembly routines called by C programs have to follow certain conventions outlined in [19]. The following calls are used to invoke the four communication routines from within a C program:

- "Qfull(i)": The destination processor is specified by index i.
- "xmitbyte(i, byte)": Destination processor index and the byte to be transmitted have to be specified.
- "Qempty(i)": The index of the source processor, from which a transmitted byte is to be read, is specified.
- "getbyte(i)": Only the index of the desired source processor has to be specified.

Thus, for sending a byte to a certain destination processor, a check is done first to determine if the transmit queue of the transceiver connected to the desired processor is full or not ("Qfull(i)"). If the queue is not full, the byte is transmitted ("xmitbyte(i, byte)"). To read a byte received from another processor, a check is conducted to determine if the corresponding transceiver's receive queue is empty or not ("Qempty(i)"). If the queue is not empty a byte is read and transmitted by issuing a "getbyte(i)" call. These communication routines correspond to the ones designed for the fault tolerant operating system in the "C" based simulation testbed (see [7]).

4.3.2. Implementation Description

The implementation of the simulation of the four intercommunicating central processors and their test software is shown in Figure 4-1 and described in detail in Appendix B.3 and in a "readme" textfile in the corresponding simulation directory ("osint").

4.4. Test Results

The detailed structure of the simulated hardware used to execute the test software described in this section is given in Figure 4-3. The following results were obtained from a test simulation doing interprocessor message passing:

- The actual message transmission could be made without requiring continuous attention from the CPU by using FIFO queues for intermediate storage of messages.
- Transmission via serial bus was properly simulated by introducing appropriate delays when transmitting via parallel buses.
- A new transceiver module was created in order to perform transmission and reception independently but in a single hardware module.
- A message (6 bytes) could continuously be passed through four processors connected in a closed loop.
- It took over a minute of CPU time on the VAX/11-780 to pass one byte from one processor to another one in the N.mPc simulation environment.
- A mechanism interfacing hardware independent high level software to the fault tolerant hardware architecture was investigated and used to implement interprocessor message passing on four fully interconnected 8086 CPUs.

The results listed above can be verified in the listing of an interactive session involving the simulation of the four fully interconnected central processors (see Appendix B.4).

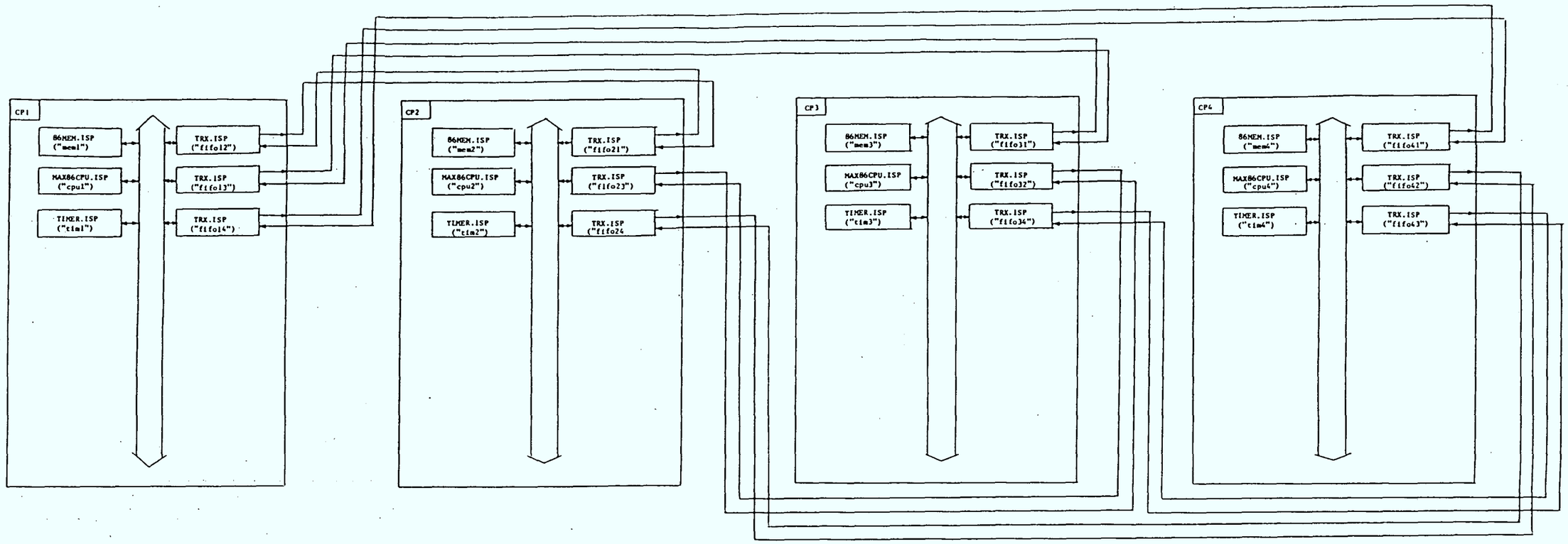


Figure 4-3: Topology of the Fault Tolerant Operating System/Hardware Integration Simulation

Finally a remark concerning the interprocessor message passing is in order. From the processor interconnection scheme shown in Figure 4-3 it can be concluded that a processor has to know its own "identity" (its position in the interconnection scheme) in order to determine which I/O port to choose when transmitting to or receiving from a certain other processor. A special CPU identity distribution mechanism had to be included in this simulation's initialization module ("newidgen.isp"). It allows a CPU to obtain its own identity (numbers 0 to 3) by executing an input instruction using a special I/O address ("IN ax, 0a0H"). The communication assembly routines then take a processor's identity and the interconnection scheme into account and are able to choose the transceiver that is connected to the desired source/destination processor. This makes it possible to simply specify the source/destination processor when calling a communications routine from within a C program. These details are also explained by the comments in the listings of the communication assembly routines in Appendix B.1.

5. SUMMARY AND CONCLUSIONS

The following hardware modules have been simulated in the hardware description language of N.mPc:

- interface processor guardian ("initipg.isp")
- peripheral processor guardian ("initipg.isp")
- gate ("gate.isp")
- initialization modules for the different simulations ("idgen.isp", "ftidgen.isp", "newidgen.isp")
- timer ("timer.isp")
- The 8086 CPU ("max86mem.isp") and memory ("86mem.isp") were designed in the course of previous simulation work.

By creating descriptions of specific interconnection schemes (topology files) and integrating the interconnected hardware modules with appropriate test software, the following, 8086 processor based, partial simulations of a fault tolerant multiprocessor architecture were completed:

- A simulation testing each one of the fault tolerant hardware modules designed in the course of this work.
- A simulation of the full fault tolerant hardware interfacing the central (high level) processors of the fault tolerant multiprocessor architecture to their peripherals. This simulation was used to demonstrate the fault tolerant multiprocessor system's ability to continue correct operation in the presence of selected, induced faults. System reconfiguration around a bus failure was also simulated successfully.

- A simulation integrating the four fully interconnected high level processors of the fault tolerant hardware architecture with the message passing routines which consistute part of the fault tolerant operating system.

The following software modules were written to run on the various simulated Intel 8086 CPUs used in the simulations mentioned above:

- Assembly modules testing each hardware module and the operation of the full fault tolerant hardware.
- Hardware dependent (assembly language) routines of the fault tolerant operating system handling interprocessor communication. These routines constitute an interface between the fully portable, hardware independent, modules of the operating system software and the fault tolerant multiprocessor architecture.
- Hardware independent high level routines that handle interprocessor message passing by calling the hardware dependent communication routines listed above. These routines were written in the C programming language.

When comparing the performance of the fault tolerant multiprocessor system described in this report to the performance of a system without fault tolerant features, we observe that the fault tolerant features add considerable redundancy and overhead due to continuous message exchanges and voting. Hence system throughput and processing power are reduced proportionately. This reduction is the penalty normally paid to gain fault tolerance capabilities. As well, the added mechanisms of synchronization and processor coordination require a careful design and verification effort.

Although publications of previously developed fault tolerant systems such as SIFT and FTMP give indication of simulation work conducted in the early stages of development to test the functional concepts of each system, no simulation details or results have been reported. We are thus unable to provide comparisons between the performance of the system described here and the systems reported in the open literature.

The simulation approach to hardware and operating system software design proved to be useful in the course of this work. Several times it has been necessary to add a new feature to the guardian initialization mechanism, change the interconnection of some hardware modules or complete the description of a bus interface which the simulation proved to be inaccurate. All these changes would have been extremely time consuming and costly if they had to be done on a real prototype hardware. The inherent flexibility of the simulation approach allows us to evaluate several design alternatives of a hardware system in a short time. This allows the designer, when he finally commits himself to implementing a certain design, to rule out early conceived options that proved to be incorrect or inefficient through the simulation work.

Concerning the adequacy of N.mPc as a design tool for multiprocessor systems several points can be made:

- In the course of this work the CAE tool N.mPc clearly proved to us useful as a hardware design and simulation tool. A complex multiprocessor system could be simulated and tested in a relatively short period of time.

- Frequent design changes to the simulated multiprocessor architecture showed the flexibility of N.mPc as a hardware design and simulation tool.

On the other hand, several limitations to N.mPc have been identified which tend to severely reduce its utility in the simulation and testing of fully integrated multiprocessor systems:

- N.mPc's slow execution speed results in a prohibitively high demand on the host CPU time if the test software modules are of substantial size. This was the case for the fault tolerant operating system.
- In the N.mPc simulation conducted here, the main (control) processors were each represented by Intel's 8086 processor. A full description of this processor is included in the library of N.mPc. The 8086 version within N.mPc was developed based on the available 8086 VLSI chip details. Like the case with many sophisticated processors, the commerically available VLSI description is not guaranteed to be complete nor absolutely accurate (bug free). All unidentified faulty attributes in the description will thus be propagated to any simulation which uses the library copy of the processor description. This complicates the process of tracing the sources of bugs when high level operating system software modules are tested in the simulation.
- The fact that N.mPc simulates the hardware down to the register transfer level is useful when newly designed hardware modules are being tested. However, when the focus of the simulation shifts to higher levels of structure modules, N.mPc

still simulates every register transfer in every microprocessor involved in lower level instruction executions resulting in a large simulation overhead. This aspect was encountered when an initial attempt was made to run the relatively complex software of the fault tolerant operating system on the fault tolerant multiprocessor architecture.

The next generation of CAE tools is expected to be endowed with top-down design and simulation features to allow the designer to follow a methodology in which he can test the lower level modules of the operating system down to register transfer level of details. Higher level modules can then be simulated and tested with the already tested lower levels replaced by macro instructions or functional blocks. This will enhance the simulation performance by several orders of magnitude over what is currently attainable by a tool like N.mPc. It will thus make it possible in practice to simulate a sophisticated system such as the fault tolerant architecture reported here in its full fledged configuration in a reasonable period of time and using modest computing resources.

The fault tolerant architecture simulated in this work is useful in many practical applications requiring continuous unattended operation. In addition to spacecraft on-board processing, these applications include computer communication switching gear, nuclear plant monitoring and processing, data collection and distribution in environmental and resource management applications and in on-site processing of remote sensing data.

A new version of N.mPc, called N.2, has been introduced by the vendor and will be available on the VAX/VMS environment. The new

version N.2 incorporates a few features aiming towards making the original N.mPc more powerful. These features are summarized below while their details are available in [16].

1. ISP′ now supports a better handling of the port constructs. Ports are N.mPc entities which allow various modules to communicate with each other. Ports are now treated as three-state devices, thus more closely resembling the behaviour of the real hardware. Capabilities for wired-OR and wired-AND have also been added.

2. Facilities for hierarchical descriptions in the ecologist have been added. This was achieved through the use of composites which are meant to be complete and already debugged simulations. A composite can therefore be considered a stand alone hardware module emulation and it can be given its own pseudo-ports which then become the only means of communications with the composite. This capability is going to be very useful as many hardware descriptions are based upon different levels of details. For example, a composite may be made to represent an Intel 86/12 single board computer and be used as a single entity even though it is, itself, made out of several ISP′ modules.

3. The Ecologist has been given the flexibility of dealing with a new hardware description language for programmable logic arrays. This should prove particularly useful in cases where the ISP′ language does not lend itself well to the modelling of gate behaviour. The Ecologist will also display the topology file(s) in a graphical format to help the designer visualize the system as it exists within the simulation

environment.

4. Better fault testing mechanisms have been provided. In particular, mechanisms to handle "stuck at" faults, "state insertion" faults, etc., have been introduced.

The enhancements made to N.mPc will offer only marginal improvements in the limitations of the package with respect to the simulation of fully integrated systems. It is felt that substantial changes in the structure of the package will be needed to make it applicable to a true top-down design approach.

6. RECOMMENDATIONS FOR FUTURE WORK

Based on the work done so far the following recommendations for future work are made:

- It is essential to monitor the technology and the availability of the next generation of CAE tools. Future CAE tools should not only be able to do hardware simulations on the register transfer level but should also include the capability of simulating complex hardware modules as "black boxes". In this manner simulations could be moved to higher functional levels while minimizing the demand for computer time for subsequent simulations.
- The hardware modules, their interconnection and the fault tolerance mechanisms should be implemented as a hardware prototype in order to fully test the integrated system capabilities when subjected to actual hardware and software failures and faults.

REFERENCES

[1] Ordy, G.M., "N.mPc: Runtime User's Manual," Department of Computer Engineering and Science, Case Western Reserve University, 1979.

[2] Ordy, G.M. and Rogers, L.A., "N.mPc: MetaMicro User's Manual," Department of Computer Engineering and Science, Case Western Reserve University, 1979.

[3] Rogers, L.A., "N.mPc: Linking Loader User's Manual," Department of Computer Engineering and Science, Case Western Reserve University, 1979.

[4] Ordy, G.M., "N.mPc: Ecologist User's Manual," Department of Computer Engineering and Science, Case Western Reserve University, 1979.

[5] Leffler, S.J., "PP: A Post-Processor for N.mPc," Department of Computer Engineering and Science, Case Western Reserve University, 1979.

[6] Rogers, L.A., "A Generalized Linking/Loader for the Allocation of the Code in Vertical and Horizontal Machines," Master of Science Thesis, Department of Computer Engineering and Science, Case Western Reserve University Report CES-79-6, August 1978.

[7] Boucouris, S., "Design and Implementation of a Fault Tolerant Multiprocessor Operating System", a Report prepared by Intellitech, March, 1985.

[8] Streit, M., "Simulation of the SBP 9989 Microprocessor Using the Computer Aided Engineering Tool N.mPc on a VAX 11/780", a Report prepared by Intellitech, September 1984.

[9] Parke, F.I., "An Introduction to N.mPc Design Environment", Proceedings of the ACM/IEEE Design Automation Conference, June 1979.

[10] Rose, C.W., Rogers, L.A., and Straubs, R.V., "The N.mPc System Description Facility," Proceeding of ACM/IEEE Design Automation Conference, June 1979.

[11] Hewitt, D.C., Parke, F.I., and Rose, C.W., "The N.mPc Runtime Environment," Proceedings of the ACM/IEEE Design Automation Conference, June 1979.

[12] Hewitt, D.C., "The Runtime Environment for N.mPc, An Adaptable System to Support the Development of Microprocessor-Based Systems", Master of Science Thesis, Department of Computer Engineering and Science, Case Western Reseve University Report CES-79-7, January 1978.

[13] Jiang, W., "A Distributed Kernel Runtime Environment for Large N.mPc System Simulation", Master of Science Thesis, Department of Computer Engineering and Science, Case Western REserve University Report CES-82-7, August 1982.

[14] Boucouris, S., "Design and Analysis of Fault Tolerant Architectures for Multi-Microprocessor Systems, Intellitech Technical Report, October 1984.

[15] Ordy G., "N.2 ISP´ User´s Manual", January 1984.

[16] Mahmoud, S.A., "VAX 11/780 CAE Tools for Multiprocessor Simulation - N.mPc Detailed System Description", September 1984.

[17] Straubs, R., "ISP´ User´s Manual", 1978.

[18] "Introduction to N.mPc System Programs", Technical Report, Case Western University, 1980.

[19] Lantech Systems Inc., "8086 C Cross Software Tools", 1983.

[20] Streit, M., "Validation of N.mPc/N.2 Microprocessor Simulation", Intellitech Technical Report, September 1984.

[21] Ordy, G., "N.mPc under VMS-Preliminary Paper", 1984.

[22] Ordy, G., "A Simple VAX N.mPc Post Processor", January 1984.

[23] Laferriere, C., "N.mPc and its Utility for Spacecraft Applications", Intellitech Technical Report, January 1983.

[24] Streit, M., "VAX 11-780 CAE Tools for Multiprocessor Simulation: N.mPc User´s and Application Manual and Installation Guide", a Report prepared by Intellitech, September 1984.

All Appendices are included in a second volume.

intellitech
Intellitech Canada Ltd
352 MacLaren Street,
Ottawa, Ontario
K2P 0M6
(613)235-5126