The Design of a Real Time, Fault-Tolerant, Multiprocessor System.

Volume 1 of 2.

Timothy Edwin Sharp.

Submitted for Ph.D. Degree.
The Computer Centre,
The University of Aston in Birmingham.
June, 1983.

The University of Aston in Birmingham

Summary

The Design of a Real Time, Fault-Tolerant Multiprocessor System.
Thesis submitted for Ph.D. Degree, 1983.
Timothy Edwin Sharp.

It is essential that real-time computers should be reliable. The majority of methods used to achieve fault tolerance in such systems employ a substantial duplication of hardware. This thesis suggests an alternative approach by placing a greater emphasis on firmware.

It is shown that a greater degree of control can be obtained in a microprogrammed computer. Furthermore, this control can often be maintained after a component failure. The use of bit-slice components is proposed as a suitable medium for the implementation of such a microprogrammed, fault-tolerant system.

It is also suggested that it is useful to overlay a high level language onto the microcoded system. A suitable language, Concurrent Pascal, is outlined. The architecture of the bit-slice processor, which has been built and tested, is described.

A set of tests, performed at microcode level, to diagnose a fault are proposed. It is shown that these tests depend upon the cooperation of another error-free processor within the system. The special problems which occur when running microcode on a faulty processor are also discussed.

The final chapter concludes that the use of microcode to achieve fault-tolerance can reduce the amount of hardware required. Suggestions for further research are also included.

Keywords: Bit-slice, Fault-Tolerance, Concurrent Pascal.

To Pam, Mum and Dad.

Acknowledgements

First and foremost, I should like to thank Dr. Malcolm Walker, my supervisor. The fact that he guided me through all the important stages in this project, whilst still allowing me to conduct the research in my own way, is much appreciated. Also, I should like to thank Neil Toye for attempting to restore sanity to some of the computing facilities and nearly always succeeding. I should like to thank Geoff Clarke for wire-wrapping some of the hardware. Finally, there are many people to whom I am grateful for their spiritual guidance, but particularly John Runchman and Steve Jamieson.

List of Contents

List of Diagrams

List of Tables

Chapter 1

Introduction

1.1 Fault-Tolerant Computing

Today, the applications of computers in real-time systems cover a multitude of uses, especially since the advent of microprocessors. Computers are entrusted with very important tasks. These include process control applications in power stations, chemical plants and oilfields and electronic applications in telephone switching systems. Also, computers are used in space systems for controlling satellites and deep space probes.

All of these real-time systems share one basic requirement. It is vitally important that the control be reliable. In other words the computer controlling them must continue to function correctly at all times. In some cases this may be essential to avoid halts in production. In spacecraft control a computer fault could mean the failure of the entire mission. More importantly, in many cases it could mean the loss of human life. It is for this reason that much computing research has been aimed at producing fault-tolerant computers ([1.1] and [1.2] for example). In line with any other field of research, once a goal has been achieved further research has been conducted to try to improve the original developments.

Several different approaches have been adopted to achieve reliability in computer systems. These approaches have varied, especially since the requirements of different fault-tolerant systems vary. For example, a telephone switching system could easily be repaired within, say, twenty-four hours, whereas a deep space probe may have to work for up to ten years without any maintenance. All of these approaches share one thing in common; to achieve fault tolerance there must be redundancy in the hardware. In other words extra hardware units must be added at some level so that if one unit fails then the others will continue to operate.

There have been two basic approaches to this problem. The first is termed dynamic or standby redundancy. This is where an outward approach is adopted. No attempt is made to improve the reliability of an individual processor or memory module. Processor units and memory modules are duplicated and some of these units remain on standby ready to take over if other units become faulty. The second approach is known as static or masking redundancy. The philosophy here is to adopt an inward approach. An attempt is made to improve the reliability of a single processor by duplicating its individual constituent components.

The following two sections in this chapter will discuss these two approaches. Following this, a common weakness of these methods is suggested and an alternative is proposed which will help to remedy this problem. Finally, in this chapter, the general aims and objectives of this project are presented.

1.2 Dynamic Redundancy

When dynamic redundancy is employed a single computer processor is broken down into three essential parts; the processing unit, the memory and a bus to communicate between the two (Fig. 1.1). Dynamic redundancy makes no attempt to improve the reliability of a single unit. However, all of these modules are duplicated. This reduces the probability of a catastrophic failure since if one component develops a fault then the system as a whole continues to function. Furthermore, the probability that two identical units will develop a fault during any given period is far less than the probability that one such unit will be faulty. If many more redundant units are added on to the system then multiple failures can also be accommodated. There are two classes of dynamic redundancy; tightly-coupled systems and loosely-coupled systems.

1.2.1 Loosely-Coupled Systems

A loosely-coupled system is one in which complete processors are duplicated and operate with a degree of independence from one another. Each processor has its own private memory which cannot be accessed by any other processors although they would have some means of inter-communication.

The simplest form of loosely-coupled system consists of a dual processor architecture. One processing unit works in main mode and the other operates in backup mode [1.3]. The main processor performs all the functional tasks required. Whenever an alteration to any vital system data is made the backup processor will be informed. Therefore, the backup system keeps a complete record of all the data stored on the main processor. This means that if the main processor fails, then the backup unit can assume responsibility and the system as a whole will suffer no loss of continuity or availability.

However, this system is very limited since if a fault of any kind develops then half the total processing power is immediately lost. Furthermore, it may take an appreciable time to repair the fault. There is no backup facility during this period. It is quite possible that the one remaining operational processor could develop a fault and thereafter no service would be available.

It is for these reasons that more recently developed loosely-coupled systems have used more than two processors [1.4,1.5]. The way in which they are linked together can vary considerably. If there is a reasonably large number of processors in a system then it will be impractical to connect every processing unit to every other one. Generally, therefore, some subset connection scheme is adopted whereby there is a limited number of links. Not all processors would be able to communicate directly with each other. Fig. 1.2 shows a typical example.

Needless to say, in such systems additional software is required to control the processor interaction and make them appear to higher levels as one complete unit.

One advantage of a loosely-coupled system is that if a processor does fail then it will not have access to memory other than its own. This helps to prevent propogation of errors. On the other hand, it

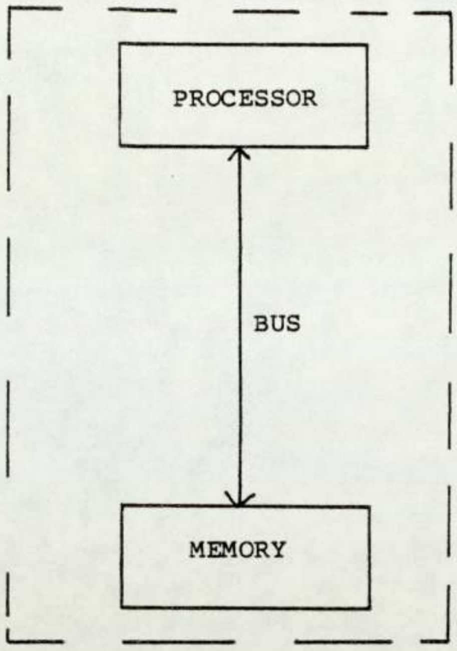


Fig. 1.1. Breaking a computer processor up into three conceptual parts.

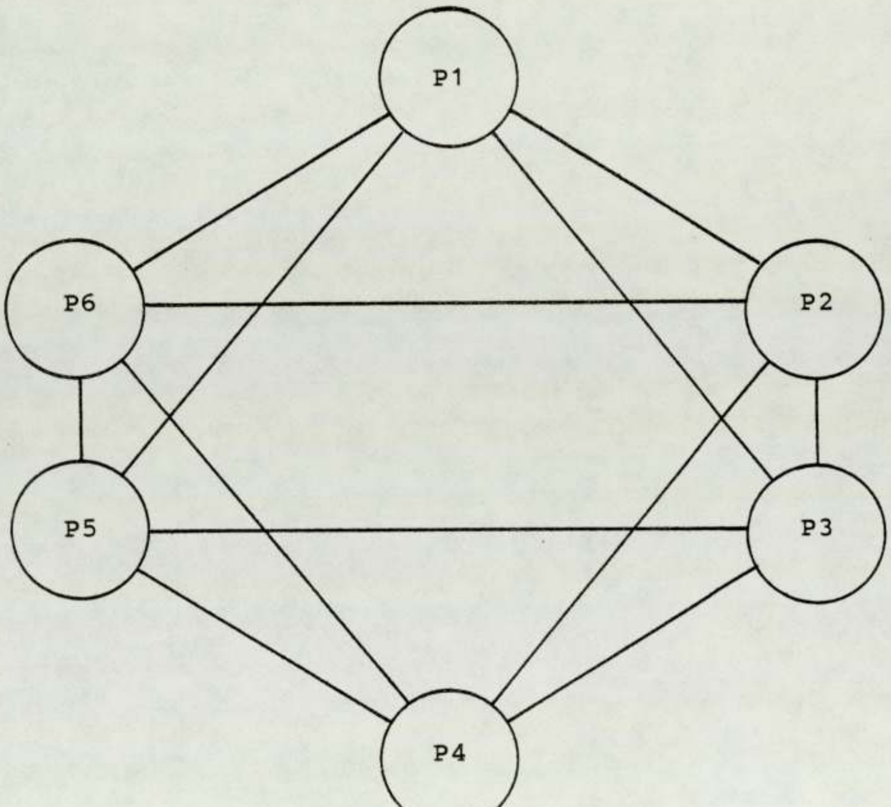


Fig. 1.2. An example of a loosely-coupled multiprocessor connection scheme.

could be argued that since a processing module consists of a processor and a memory then the fault will have occurred in either, but not both, of these. This means that the complete processing unit, which contains either an operational processor or an operational memory, is disabled. Therefore, in hardware terms it could be argued that a fault causes twice as much damage than necessary.

1.2.2 Tightly-Coupled Systems

The second way that dynamic redundancy can be achieved is by means of a tightly-coupled system. This is where a multiprocessor consists of a common pool of processing units and a shared set of memory modules. All processors will have access to any memory module. The essential difference between a tightly-coupled system and a loosely coupled one, therefore, is that the former has shared memory and the latter does not.

The most common architecture employed within a tightly-coupled system is one in which processors and memory modules are connected by duplicated shared buses. Fig. 1.3 gives an example of such a scheme, although many variations are possible [1.6].

A failure of any one unit (this includes a bus failure) means that only a small part of the processor is out of service. However, there are hardware problems since the memory modules must have a separate port for each bus, i.e. a multi-port memory is required. Also, there is a very real danger of a faulty processor "running wild" and corrupting memory which is being used by other units. It

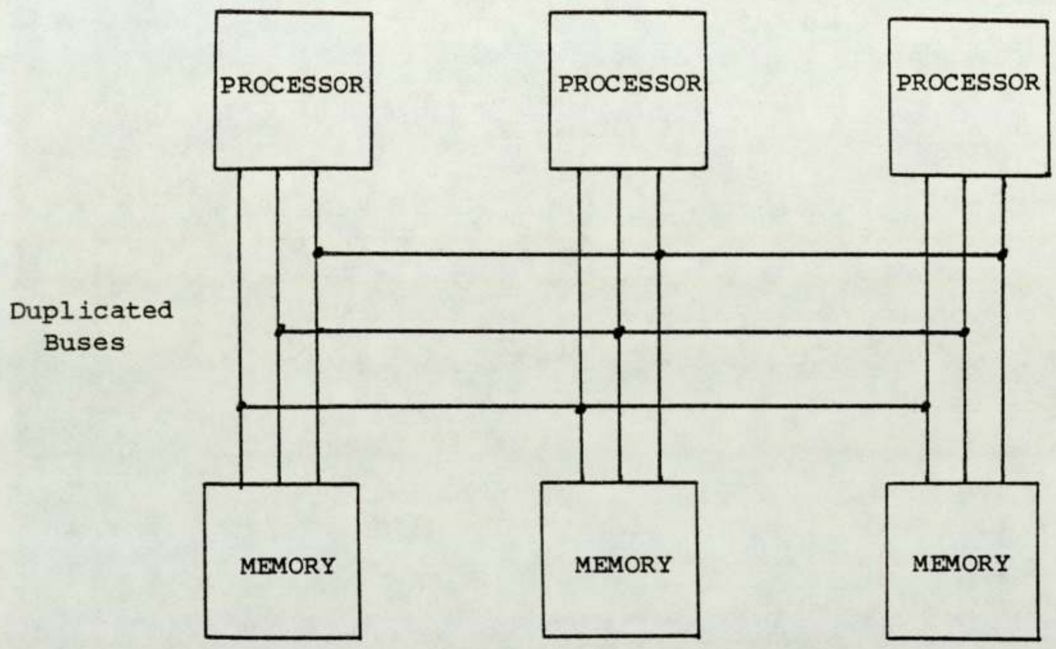


Fig. 1.3. An Example of a Tightly-Coupled Architecture.

is for this reason that hardware protection registers are employed. A processor would have to access a memory module via one of these registers. The registers would contain preset upper and lower addresses and they would not allow a processor to access an area of memory outside this range. In other words each processor has a section of memory which it is allowed to use and this protection is enforced by hardware registers. Tightly-coupled systems thus require extra hardware. On the other hand, they require less memory than loosely coupled ones.

As with loosely-coupled systems there is an additional overhead in terms of the amount of software needed to control the multiprocessor. There are many control schemes. The processors may be regarded as an anonymous pool of resources as in the Plessey System 250 [1.7, 1.8, 1.9] or they may each have separate tasks to perform as in the JPL-STAR [1.10]. Some systems use a combination of tightly and loosely coupled schemes so that there will be a pool of shared memory but each processor will also have a private store [1.11].

1.3 Static Redundancy

An alternative approach that has been taken to achieve fault-tolerance is to make single processors more reliable by utilising intra-processor redundancy. Rather than duplicating processors an attempt is made to maximise the internal reliability of a single processor. There are two types of static redundancy; Triple Modular Redundancy (TMR) and Self-Checking checkers.

1.3.1 Triple Modular Redundancy

If a computer is built using Triple Modular Redundancy (TMR) then all vital components are triplicated. Also, a voting circuit is added to allow the three identical units to ignore the output from a faulty unit. In Fig. 1.4 there are three identical units; U1, U2 and U3. The voter chooses their majority output. Normally, if all units are non-faulty then all three outputs will agree. However, if one unit is faulty then the voter will choose whichever two outputs agree. If two or more units are faulty then the voter may choose the incorrect outputs.

Although the failure of a single unit can be tolerated a fault in the voter will cause the circuit to fail. Fortunately, the structure of a digital computer is such that most components are constantly passing values between one another. As an example take a register and an adder (Fig. 1.5). In a normal non-redundant system the register will produce one of the inputs of the adder and the results of the addition will be placed back in the register. If the register and the adder are triplicated then there will be six

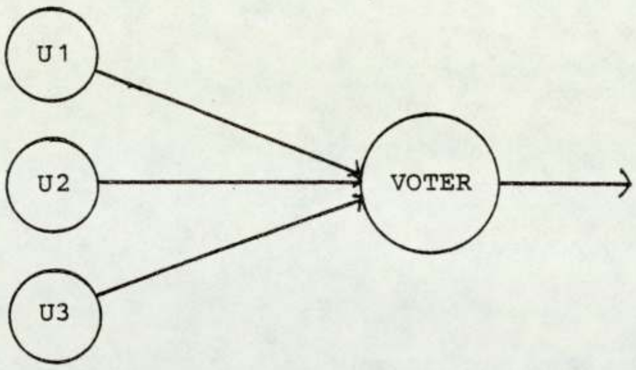


Fig. 1.4. Triple Modular Redundancy.

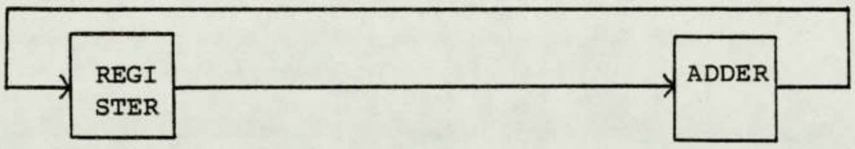


Fig. 1.5 a). A non-redundant ALU.

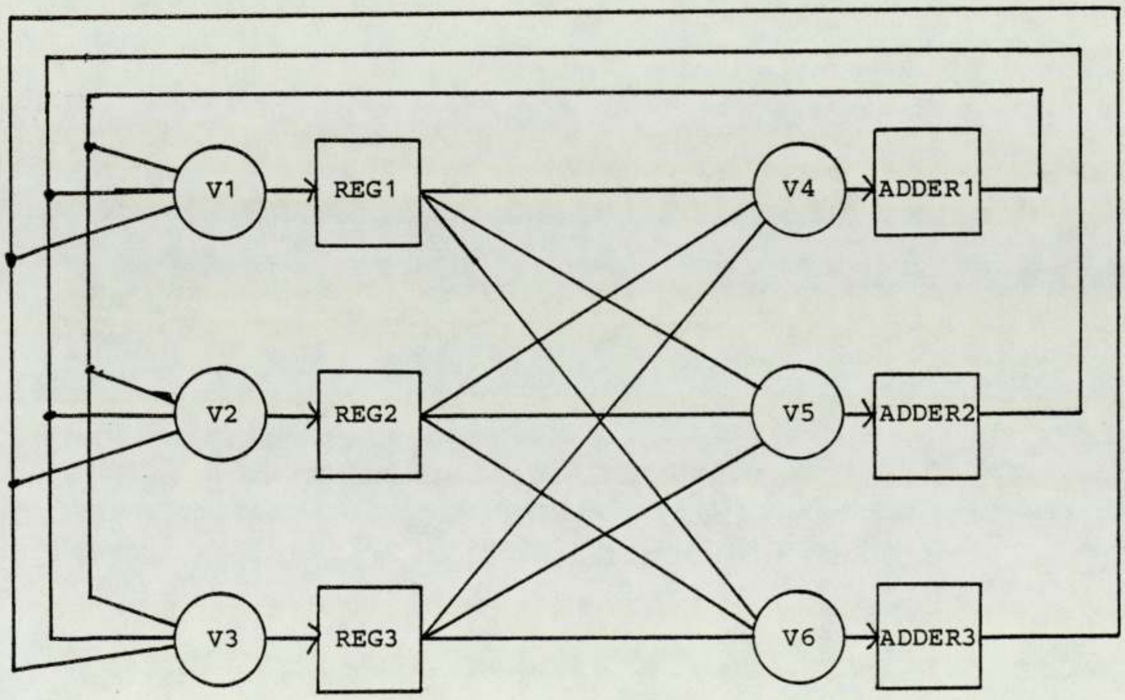


Fig. 1.5 b). An ALU implemented using TMR.

voters. Each one of these will be associated with the input to a particular unit. If a voter fails then it will have the same effect as if the unit it was feeding had failed. If, for example, in Fig. 1.5 b) the voter V1 failed then REG1 would contain incorrect data. However, this fault would not propogate since the three voters feeding the adders would mask out the incorrect data.

TMR is, in one sense, an ingenious solution since it requires no additional software and can tolerate any single fault within each sub-system. TMR can even be used at higher levels since the units which were given as examples in Fig 1.5 could equally well have been processors and memories [1.12]. TMR, however, does require an increase in hardware by a factor of greater than three. This is because voting circuitry is required in addition to triplication of units. This increase in hardware can increase the probability of a fault occurring and although the system can tolerate this, it could not withstand a multiple failure. In order to be able to tolerate two faults of any description five units of each type would be required. TMR, therefore, although requiring no extra software, requires a large increase in the amount of hardware.

1.3.2 Self-Checking Checkers

Another form of static redundancy which can be a useful aid for fault-diagnosis is in the use of self-checking checkers [1.13]. In this situation a functional unit within a processor is duplicated but only one of the two units produces an output which is used by other parts of the processor. The second redundant unit is used to check the output of the first one, using a self-checking checker. Fig. 1.6 gives an example of this using an ALU. A self-checking checker produces one particular output if the two units agree and another output if they disagree or the checker itself is faulty. The circuitry becomes more complicated when comparing 'n' bits but the manner in which it operates can be illustrated in Fig. 1.7. This shows a self-checking checker for comparing the two bits x1 and x2. It can be seen by examination that if x1 = x2 then the outputs (f1,f1') will be either (0,1) or (1,0). If the outputs (f1,f1') of the checker are either (0,0) or (1,1) then either its two inputs are unequal or the checker itself is faulty. A description of how this is implemented for n-bit comparisons can be found in [1.14].

Unlike the other methods described so far in this chapter self-checking checkers are used to assist in fault diagnosis rather than fault tolerance. This is nevertheless useful for two reasons. First, it is important to be able to detect and diagnose a fault when it occurs in order to isolate it. Second, failed components can be replaced more quickly, thus reducing the probability of multiple faults. Self-checking checkers perform a similar function to parity and error-code checks, both of which are well established and documented [1.15]. However, self-checking checkers are more reliable since they can detect faults of any multiplicity [1.14,

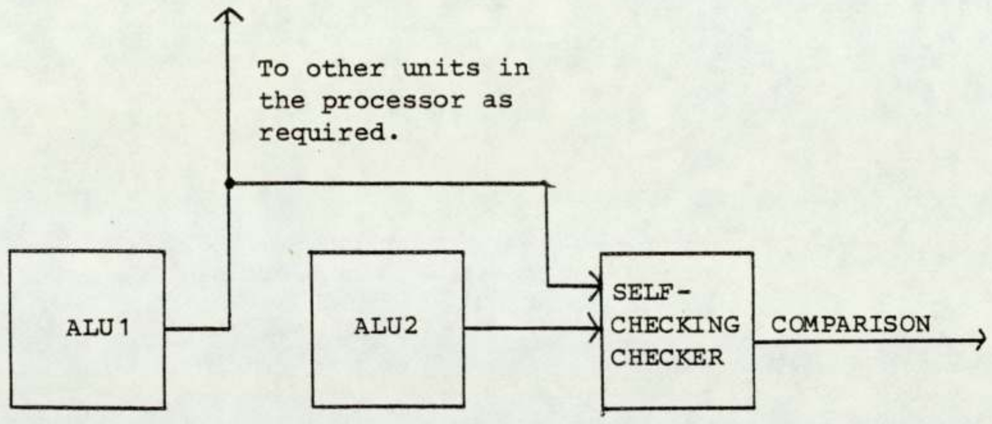


Fig. 1.6. Use of a self-checking checker to implement an ALU.

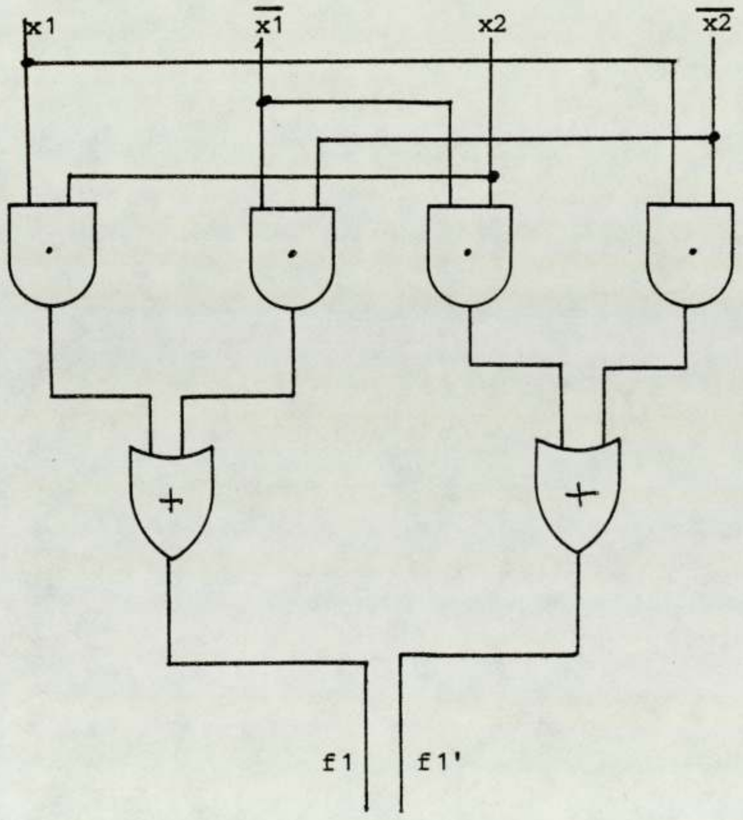


Fig. 1.7. A one-bit self-checking checker.

1.16, 1.17, 1.18].

1.4 An Alternative approach

There are many variations on the basic themes described in the previous two sections. However, it is hoped that this chapter has outlined the various methods that can be used to achieve fault-tolerance. All of the approaches discussed require a disproportionate amount of resources to achieve fault-tolerance when compared with normal systems capable of performing the same tasks without fault tolerance. These methods always require additional hardware to achieve redundancy and they may, in many cases, require special software. Furthermore, a situation is rapidly reached where the addition of further hardware produces very little increase in reliability. The author believes that some so called non-redundant systems contain a certain amount of duplication. This can be regarded as a form of redundancy if used correctly. If this were exploited fully it would help to reduce the amount of redundancy required at other levels using the methods that have been described. This section will attempt to outline the rationale behind this statement.

Consider an ordinary computer processor which is made up from thousands of components. A failure of any one of these will render this processor useless although the vast majority of its components are still operational. Since these components are mutually interactive and dependant it would be impossible to isolate a fault at this level. However, if a processor is regarded as being split up into a few separate subsystems then the same argument can be applied. A processor could, for example, be regarded as being split up into a memory module, a bus, an arithmetic unit and a control unit. If the memory bus were to become faulty then the processor as a whole would be inoperable. However, the processor, consisting of a control unit and an arithmetic unit would still be functioning correctly. Once it is unable to access memory to obtain the instructions the complete processor becomes inoperable.

In the past it has been practical to regard a computer system as being made up of two essential elements; hardware and software. An additional, third element is used in many digital systems; firmware.

If a computer processor employs a microprogrammed control structure [1.19] then it will be arranged as follows. All sub-systems within the processor such as the ALU, the memory, etc. will be directly under the control of the Microprogram Control Unit (MCU). The MCU consists of the microcode memory together with some means of sequencing the accesses to this memory (the Microcode Sequencer). Each word of microcode memory consists of a series of bits (ones and zeroes) which will be routed directly to the sub-system they are controlling. Since there can be many sub-systems with several functions a fairly large number of control pulses are required. The length of a word of microprogram memory can be quite large when compared to main memory (usually between 64 and 128 bits). However, the number of words required is comparatively small (typically between 1/2K and 4K). Consequently, the addressing mechanism tends to be fairly simple.

There are two advantages to a structure of this type when used with fault-tolerance. First, since the MCU is relatively simple in terms of its addressing mechanism it is less likely to develop a fault than other functional units. Second, if the main memory does become inaccessible then the MCU offers an alternative source of control. This source is located entirely within the processor itself. Consequently, it may be able to make use of the sub-systems within the processor that are still operational.

Some commercially available computers already employ a technique known as micro-verification. This involves checking the hardware status of the machine at microcode level. One example of this is the current PR1ME range of minicomputers [1.20]. The microverify routines on the PR1ME consist of a set of 11 tests. These are always executed at the start of the power-on sequence. They can also be explicitly called by the operator at the control console and they are also called in the event of a processor parity error being detected. If a test fails then the machine will halt and the number of the failed test can be displayed on the front panel of the machine (assuming it is still operational).

These tests consist of passing values through various registers and of testing buses, ALU functions (add, subtract, shift etc.), parity, the I/O bus, and main memory. These tests prevent a faulty machine from being put into operation after being powered up, thus preventing it from corrupting vital data in areas such as the backup store.

1.5 Aims and Objectives

Microcoded control can take place to a limited extent under degraded conditions on commercially available computers. This indicates that it is a viable proposition to investigate its further use in fault-tolerant systems. It would be useful to analyse thouroughly the structure of a microcoded computer to see how its architecture can be used to obtain a level of fault-tolerance. The limited scope of a project such as this means that the amount of time available is small. It is for this reason that two specifications of the research to be carried out should be as follows. First, only one existing computer architecture will be selected and analysed rather than proposing a general theory of fault-tolerance. Second, the research, when completed, will be in such a state that it will be easy for subsequent researchers to continue the work.

After a processor architecture has been selected and analysed a basic fault-tolerance philosophy for this type of microcoded machine will be derived. Following this, an attempt will be made to implement it. It is felt by the author that this is important for two reasons. First, since little work has been done in this field it would be prudent to prove by experimentation that any principles derived are correct. Second, it could be useful for future researchers to have access to a working system. This could be used as a vehicle for further research.

There are two ways in which the proposed research could be carried out. The system can either be simulated or the appropriate hardware can be built. Simulation is a powerful tool. However, it was rejected for the following reasons:-

1) Simulation requires powerful computing facilities. It was felt that the level of detail involved in such a simulation would result in considerably more computing power being required than was available.
2) The structure of any processor, chosen for simulation, would involve several layers. In addition to the hardware levels there would need to be simulation of the microcode, machine code and (possibly) the high level language. This could be a complex structure to simulate.
3) It was felt that considerable benefit would be gained by designing and implementing a processor, rather than merely simulating the structure.
4) This processor would then be available for future work.

A means is required whereby a fault-tolerant computer system can be built economically, but which uses an architecture that is specific to requirements. One of the developments in commercial microprocessors in recent years has been the advent of bit-slice computers. Bit-slice elements allow a computer designer to implement a processor from individual sub-systems. This can be done using commercially available devices rather than buying a complete "off the shelf" processor. In order to be able to define the architecture of a processor completely the designer needs to be able to specify the machine instruction set. Consequently, a microprogrammed control structure is an inherent requirement of a bit-slice processor. This would, therefore, make an ideal vehicle on which to conduct the proposed research.

Some research has already been conducted in this area [1.18]. The approach that will be adopted here will vary from this earlier work. First, the research already carried out used some additional hardware. Second, a faulty sub-system attempted to analyse itself by using its own internal components. The approach described in this thesis allows a faulty sub-system to be analysed by external functional units. Therefore, the difference between these two approaches is that the fault diagnosis will operate at different levels. More importantly, an attempt will be made to maximise the degree of fault tolerance available from an existing architecture without using any additional hardware.

The rest of this thesis describes the work that has been undertaken. Chapters 2 and 3 describe the decisions that were made in choosing a suitable design and also describe the system itself. Chapter 4 describes the fault-tolerant principles derived and Chapter 5 outlines the problems that were found when an attempt was made to implement them. Finally, a conclusion is presented in Chapter 6.

Chapter 2

Concurrent Pascal

2.1 Advantages of a High Level Language

An important decision that had to be made at the beginning of this research project was whether the multiprocessor, when built, should have the ability to support a High Level Language (HLL). Since the hardware was to be built from Bit-slice components the system programming could take place at either HLL, machine code or microcode level (Fig. 2.1). A HLL programming capability does not diminish the ability to program at other levels.

As one moves from a high to a low level there is a decrease in software productivity and an increase in the amount of detailed machine knowledge required. The advantages of programming in a HLL are obtained at the expense of code efficiency. The amount of time taken to process any given algorithm written in microcode is significantly shorter even than in machine code but a detailed knowledge of the processor architecture is needed. At machine code level it is usual to require only an awareness of the register transfers that take place within the computer. However, machine code is still one and a half to two times faster in execution than a good real-time language. If a HLL is used then virtually no knowledge of the machine architecture is required. All that might be necessary are any specific details that are peculiar to the language implementation on the particular machine. It will be shown in Section 2.3 and Appendix 11 that, in the solution eventually adopted, these are negligible. The one level of software which

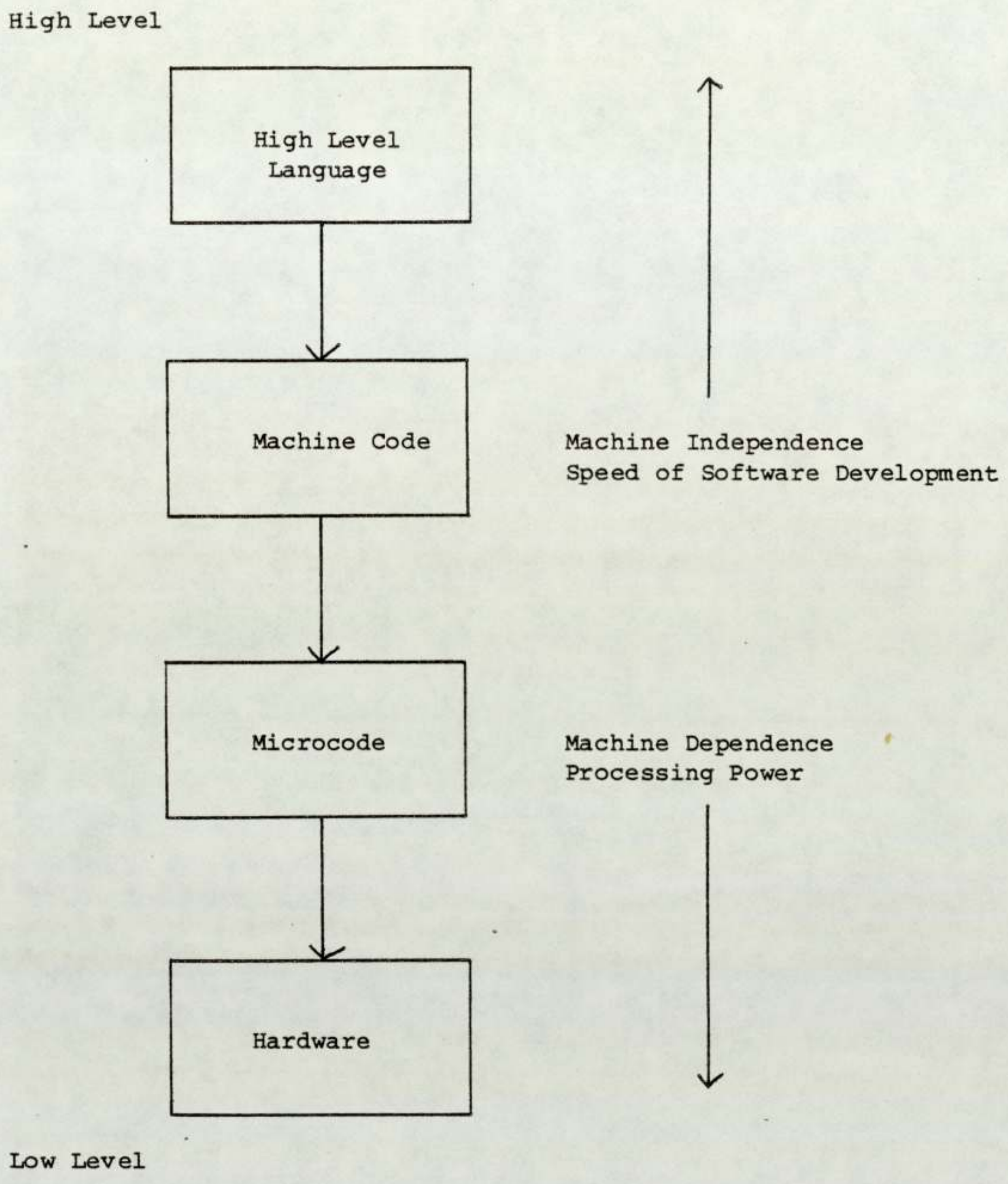


Fig. 2.1. The three levels of Software Development.

could not be entirely eliminated is the microcode level. This is because whatever permutation of the three levels is used they must be implemented in firmware.

If a HLL is implemented in microcode (Fig 2.2), as opposed to machine code on a conventional non-microprogrammable processor, it can be more efficient. This is because an optimum machine instruction set can be chosen as the compiled and executable form of the code. Furthermore, once the basic firmware implementation of the HLL has been achieved it would be possible for much of the development to be performed at a high level. This would make it possible for any future researchers to continue with the project and program the computer with no knowledge of the machine architecture being required. The use of a HLL, therefore, would give an enhanced rate of code development with the first steps being undertaken in microcode and the latter stages using a HLL. The choice of whether to use machine code need not be taken until the HLL has been implemented. Once a HLL capability has been attained the ability to program in machine code still remains. However, the use of machine code is at the software designers discretion.

The use of microcode at low level can be used to advantage in providing fault-tolerant test routines. If a set of fault-tolerance routines are imbedded in the firmware and can be called from higher levels then the effectiveness of these routines can be efficiently evaluated. This can be done, for example, by calling them at different times and in different orders. Again, this is merely an option while the HLL is being developed. The software designer is still able to write code at the microprogram level after a HLL has been implemented.

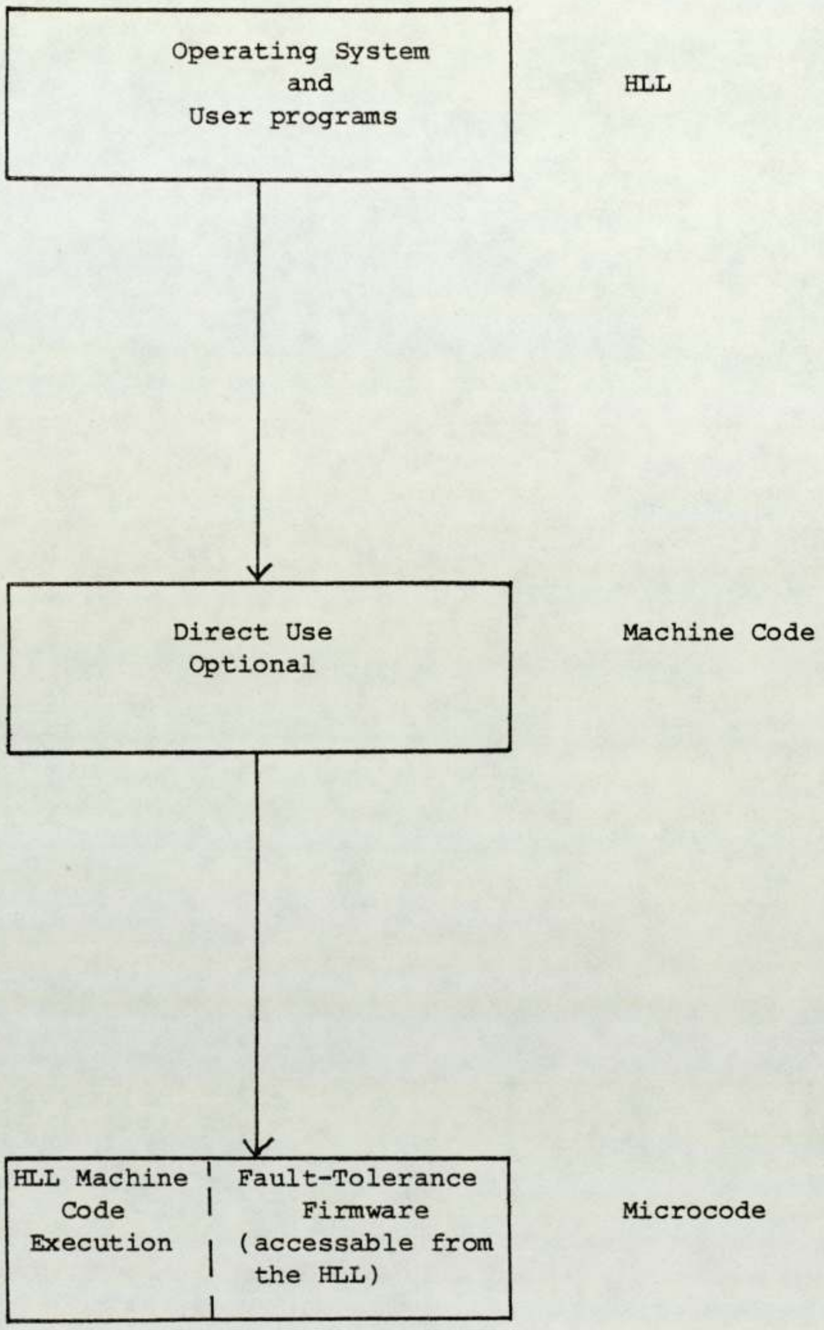


Fig. 2.2. Implementation of a HLL.

To summarise then, a HLL implemented in microcode minimises the loss in processing power whilst retaining the speed of software development attributed to such a tool. It would make the system more approachable to new researchers. Also, it would allow certain software design decisions, such as where the fault-tolerance code should appear and the expandability of the system, to be deferred until after the HLL had been implemented.

It was therefore decided that the processor should have the ability to execute programs written in a HLL. One of the main arguments which influenced this decision was that a microcode implementation could be very efficient. It was felt necessary to choose the HLL before the processor architecture was selected. This was because an architecture which could efficiently execute the HLL was required. The main indicator of whether a processor structure is suitable is the run-time machine code produced by the HLL compiler. It was therefore necessary to select a processor architecture that was well matched to the run-time machine code. This made the choice of a suitable HLL the next logical step in the research.

2.2 Languages Considered

A suitable language for this type of project would be, as the title of this thesis suggests, one designed for a real-time application. Alternatively, a systems programming language could be suitable. In order to achieve a working system within the timescales allowed, it was important to choose a language which had a readily available compiler and needed little effort to transport it. Therefore, in the end, practical considerations prevailed in choosing between theoretically equally matched languages.

Five languages were investigated for use on this project. They were CORAL 66 [2.1], RTL/2 [2.2], MODULA [2.3,2.4,2.5], ADA [2.6] and CONCURRENT PASCAL [2.7].

Coral 66 was the oldest of the languages. It has been a popular real-time language for many years, especially in industry. It was felt, however, that a more modern language with greater flexibilty should be employed.

The second real-time language that was considered was RTL/2. This was a language that was rapidly gaining popularity and was quite suitable. It was more modern than Coral in its design philosophy. There was also a good portable compiler available.

Another language that was considered was the most recently designed of the five; Ada. It was designed as a general-purpose language and incorporated facilities normally found in both real-time and system-programming languages. Unfortunately, Ada was so new that no compiler existed and, although it would have been highly suitable, it was immediately discounted.

The fourth language that was investigated was Modula which was designed as an operating systems language and was based on Pascal. The structure of the language was such that it was possible to maximise the amount of software written in Modula and minimise the amount written in machine code. Modula is a multiprocessing language and embodies the concept of concurrent processes. At the time of the investigation the author was not aware of the location of any readily available compilers.

The final language that was considered was Concurrent Pascal which was also based on Pascal. It was designed for implementing small operating systems and was therefore suitable for the application required. Again, like Modula, Concurrent Pascal is a multiprocessing language. However, the multiprocessing concepts are applied in different ways. It is beyond the scope of this text to critically compare Modula and Concurrent Pascal but the author preferred the approach to multiprocessing that the latter took. Also, Concurrent Pascal was found to be more readable. These factors, combined with the lack of availabilty of a compiler for Modula, led the author to select Concurrent Pascal out of the two multiprocessing languages.

Having eliminated Modula for the above mentioned reasons, Coral on the grounds of its age and Ada due to the lack of a compiler the author was left with a choice between RTL/2 and Concurrent Pascal. The factor which eventually swayed the balance was not the high level features of the languages but an aspect at a lower level. In order to achieve portability both RTL/2 and Concurrent Pascal use

Intermediate Level Languages (ILL). The ILL used by Concurrent Pascal, called P-code, is at a lower level than the one used by RTL/2. P-code is actually a hypothetical machine code. This means that it would not be necessary to design a machine instruction set if Concurrent Pascal were used. This would mean that there would be one fewer task to perform, thus allowing more time and effort to be spent on the fault-tolerance aspect of this project. Designing a machine instruction set to implement a HLL is difficult in terms of achieving the optimum solution. Also, P-code is based on a zero-addressing (or stack-machine) architecture which is particularly efficient for the execution of a high-level block-structured language such as Pascal [2.8].

Therefore, the use of Concurrent Pascal would ensure that by writing microcode to execute P-code an efficient implementation of the language could be obtained. This fact, coupled with the relative ease of obtaining and transporting a compiler, led to its selection as the HLL to be used. However, this choice virtually closed another design option. P-code, although very suitable for the execution of a HLL, should not be considered as a general purpose programming language. This is because the programmer needs to keep track of the current state of a frequently used stack. Also, P-code makes many more memory (stack) accesses than a normal 1 or 2-addressing machine code. These memory accesses would occur in any implementation of a block-structured language but not in software written directly in machine code. There would therefore be very little gain in processing speed obtained by writing software directly in P-code. This meant that it was only practical to write software at the HLL or the microcode level. This restriction did not seem to be particularly important for two reasons. First, an extremely efficient implementation of the HLL would be obtained. This meant that some code requiring fast processing could be written in Concurrent Pascal. Second, programs which would normally be written in machine code could be implemented in microcode which is faster.

2.3 A Brief Overview of Concurrent Pascal

2.3.1 The Language

In subsequent chapters a description of the high-level, fault-tolerance software will appear. In this section a description of the basic concepts of Concurrent Pascal will be presented. Hopefully, this will help to give the reader an understanding of the design decisions that were taken. For a more detailed description of Concurrent Pascal the reader should consult [2.9].

Per Brinch Hansen, the designer of Concurrent Pascal claims that his aims were to achieve simplicity, reliability, adaptibility, portability, efficiency and generality. The language was designed for implementing small operating systems, although Brinch Hansen has stated that it could be used for larger operating systems. This has been done to implement a time-sharing system in [2.10]. The language is based on the well-known and popular Pascal programming language. The language is structured so that the hub of the operating system is written in Concurrent Pascal and all the user and application programs are written in Sequential (ordinary) Pascal. One of the main aims of the language is to ensure that any time-dependent, and possibly non-predictable, errors caused by concurrency cannot occur. To this end, the language depends very heavily on the compiler to trap any such potential errors, thus reducing or even eliminating them.

Concurrent Pascal extends Sequential Pascal by the use of three concepts, Processes, Monitors and Classes.

A typical Concurrent Pascal program would consist of two or more processes each one merely being a piece of code. All processes would run concurrently. Thus, for example, one process might handle input, another run user programs and a third handle output.

The main problem that occurs with concurrency is that of process communication. In order to achieve this, processes must share data in memory. However, if they attempt to write to a shared-data area simultaneously then it may be corrupted. There are several means of overcoming this, Concurrent Pascal uses Monitors as a solution.

A Monitor consists of some data (in the form of VAR declarations) and some operations (in the form of Procedures) to access this data. A process that uses a Monitor cannot access its data directly, it can only call the operations that act upon this data (Fig 2.3). To ensure that there are no concurrency problems, only one process can use the Monitor at a time. If a Process attempts to call a Monitor which is already being used then it must enter a queue and await its turn to obtain exclusive access.

The way that Processes communicate to each other via Monitors can be shown diagramatically using access graphs (Fig 2.4).

Therefore, in this way, only one Process can access a shared data area at a time. However, a Process is unaware of the detailed structure of the data. For example, a Monitor might consist of a buffer (an array) together with a set of operations. There might be an operation to read an item from the buffer and another one to write to the buffer. It is because a Process is only aware of the existence of data and not its structure that Monitors are referred to as Abstract Data Types.

Because a Monitor call carries a large run-time overhead another structure called a Class is also employed. This is similar to a Monitor as it contains some data and a set of operations to act upon the data. However, it is ensured at compile time that only one Process has access to a Class. A Class therefore posseses the data abstraction of a Monitor but it reduces the run-time overhead. This concept has been adapted from the language Simula 67 [2.11].

There are also some Process scheduling facilities in Concurrent Pascal. Suppose a Process calls a Monitor to read an item from a buffer which is empty. A Monitor has the ability to Delay the Process using a Queue which is simply another variable type. When the buffer becomes full again (i.e. another Process puts data into it) the Monitor has the ability to Continue a Delayed Process.

There also exists a real-time primitive called Wait which delays the calling Process for one second, thus allowing long-term real-time scheduling.

A Concurrent Pascal program has the ability to call a Sequential Pascal program in virtually the same way as it might call a Procedure. However, it is responsible for first ensuring that the

Sequential Pascal program is ready to run. For example, by reading it in from disc into memory.

2.3.2 The Implementation

The implementation of Concurrent Pascal is somewhat different to that of conventional sequential programming languages. For example, an ordinary implementation of Pascal would only require some means of executing P-code. This would be performed either using an interpreter, a translator or by direct execution with microcode. However, in addition, Concurrent Pascal requires a means of executing its additional Concurrent P-code instructions. One possible method is to write an interpreter for the Sequential P-code and a Kernel for the Concurrent P-code. This classic approach was adopted in the original implementation and also by most subsequent users of the language [2.12, 2.13]. Brinch Hansen acknowledges that interpretation results in a loss of efficiency which can be eliminated if a firmware implementation is employed. It has already been stated that this method would be adopted on this project. To the author's knowledge this is the first approach of this type which has been attempted, although firmware implementation of ordinary Pascal has been achieved [2.13]. Also, [2.14] did adopt a compromise approach of microcoding frequently used Sequential P-code instructions.Therefore, it is felt that the implementation deserves some mention as it adds an extra element of originality to the project.

The implementation of the Sequential P-code was fairly straightforward although time-consuming and is fully documented in

Appendices 7 and 9. The Kernel is the part of the microcode that makes a single processor appear as a multiprocessor with one processing unit for each process. This is done by multiplexing the single available processor between the software processes defined in the high-level program. The Kernel is also responsible for policing Monitor usage, this is done by means of a structure called a Gate (Fig. 2.5). This consists of a boolean variable called OPEN which indicates whether the Monitor is in use and an array in which Processes can be queued. If a Process calls a Monitor the Kernel checks to see if it is in use. If it is, the caller is put in the queue. Whenever a process leaves a Monitor the next one in the queue is allowed to enter the monitor.

The Kernel is also responsible for the real-time facilities such as the Wait directive and must therefore maintain a record of the time. In addition, it is responsible for input and output.

Therefore, it can be seen that the Kernel constitutes the core of an operating system consisting of process scheduling, shared-data access policing, real-time facilities and input/output. The extra operating system features required are overlayed onto this using high-level Concurrent-Pascal code. Since parts of the operating system are written in microcode they are very efficient. The operating system interface between the high-level code and the microcode is by means of P-code instructions (Enter Monitor, Leave Monitor, Initialise process etc.). This notion of having low-level scheduling operations that can be called from a higher level was also adopted in the only other microcode implementation of a scheduler that the author is aware of [2.15].

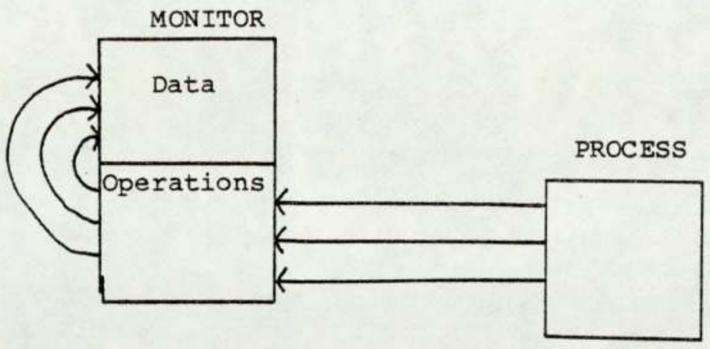


Fig. 2.3. A Monitor.

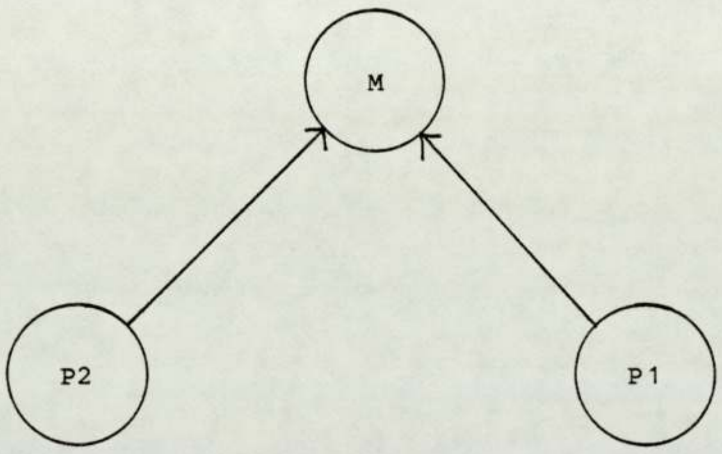


Fig. 2.4. An Access Graph of two processes P1 and P2 communicating via a Monitor M.

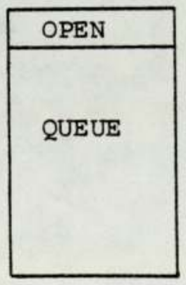


Fig. 2.5. A Gate.

Therefore, the kernel is (in this case) a microprogrammed implementation of the concurrent aspects of the language. It is the part of the implementation that tends to be altered by new users of the language in order to achieve any changes required [2.10]. It is also, in this case, an unusual approach to writing an operating system by using microcode. A detailed description of the Kernel can be found in Appendices 8 and 9.

2.3.3 Adaptability

Concurrent Pascal had been originally designed to run on minicomputers with many peripherals such as disc drives, paper and magnetic tape etc. The implementation required here was on a smaller system, as described in Section 3.2, whose only i/o was either to a terminal or to another processor. In addition, it was not possible to store Sequential Pascal programs on disc and read them into memory before execution. All code, both Concurrent and Sequential, was stored on EPROM. These were machine details that were impossible to hide from the software programmer. They affected two areas of the high-level software, namely the "io" command which was responsible for input/output and also the mechanism for calling Sequential Pascal programs. The project already required a diversified knowledge of hardware, firmware and software. Therefore, the author felt that it would not be practical to alter the Concurrent Pascal compiler as well. Fortunately, Concurrent Pascal proved sufficiently flexible to be able to alter machine details without amending the language definition.

The input/output directive, io, is called with three parameters.

As a special case the compiler performs no type checking on these parameters. It merely generates P-code to push them on the stack and make an "io" call on the Kernel. This means that it was only necessary to define the new i/o operations within the Kernel. Meaningful names can be given to the values required as parameters in the high level code using standard Pascal "type" and "const" declarations. This approach was taken from [2.10].

The calling of programs also required no alteration to the language, only its implementation had to be changed. Both of these machine-dependent modifications are defined formally in Appendix 10.

Chapter 3

Bit-Slice Hardware

3.1 The principles of a Bit-Slice architecture

The term "Bit-Slice" is derived from the fact that a basic processing element comes in a package with a small bit width. To implement a larger unit of the same description several of these devices would be cascaded together. For example, to construct a 16-bit ALU, 4 X 4-bit sub-units would be used. These devices have various inputs which control such factors as the function to be performed, the data to be operated on and where the results should be stored. For an ALU the functions would include adding, subtracting, shifting right and left and so on. The data to be operated on could come from internal registers or an external source. If the former is chosen then the register number must be specified. The outputs should also be to an internal register or an external destination. Connecting the control inputs of these devices to the microcode memory gives the microcode control unit complete command over the operation of the ALU.

One of the main manafacturers of bit-sliced products is Advanced Micro Devices (AMD) whose components were used in the design of the hardware to be described. The following descriptions draw freely on technical specifications from the Am2900 range of bit-slice devices. Further details of these components are given in [3.1]. A further description of bit-slice principles can be found in [3.2].

3.1.1 Construction of a bit-sliced ALU

A 16-bit ALU which has been constructed from 4 X Am2903 bit-slice devices is shown in Fig. 3.1. Since the four devices are meant to act as one unit they must perform the same operation on data from the same sources. The Am2903 has 16 internal registers and any two of them can be accessed at a time. Accordingly, the control pins of the Am2903 have 4-bit A and B register select fields which indicate the register to be operated on. In order to make sure that each device operates on the same registers the field select control pins on each chip are connected together. There are 9 ALU function and destination control signals. These are designated I0 to I9 and the corresponding pins on each package are connected together. I0 to I4 control the function (add, subtract etc.) and I5 to I8 determine the destination (whether to latch the result into a register, whether to shift it, etc.). All these functions are defined fully in Tables 3.2 and 3.3. The source of data is defined by the $\overline{\text{Ea}}$ $\overline{\text{OEb}}$ and I0 signals which are connected together on each device. The sources available are defined fully in Table 3.1, they include an internal ALU register or one of the two external Da or Db inputs. Typically, these would be connected to a source such as the memory data register. Each bit-sliced device would receive 4 bits of this data. Any outputs, as well as being latched into registers, would appear at the Y outputs of these devices. All of the control inputs described above are connected to the microcode instruction register. This gives the microcode complete control over the ALU.

Whenever there is a device boundary two things must happen to make the ALU function correctly :-

1) Any data which overflows in an arithmetic operation in one device must be added on to the value in the next most significant sub-unit. Accordingly, the Carry Out (Cout) output of each device is connected to the Carry In (Cin) input of its most significant neighbour.

2) Any data shifted out of one device must be shifted in to the next one. Accordingly, the shift-out outputs of each sub-unit (SIO3 = Shift Out Left, SIO0 = Shift Out Right) are connected to the shift-in inputs of its neighbour (SIO0 = Shift In Left, SIO3 = Shift In Right). This allows the ALU to perform 16-bit shifts.

The above is a simplistic description of how large units may be constructed from smaller components and how the microcode controls them.

3.1.2 A Simple Bit-Slice Architecture

A very simple bit-slice computer will consist of three basic functional units; the Computer Contol Unit (CCU), the Arithmetic and Logic Unit (ALU) and the main memory (Fig. 3.2).

The CCU consists of the microcode which is usually stored in ROM, it is not related to main memory where the machine code is contained. There must also be some means of sequencing the microcode. This entails generating the next microinstruction address, allowing for conditional jumps, subroutines and other transfers of control. The CCU is at the heart of a bit-slice computer. It directs all the other components by means of control

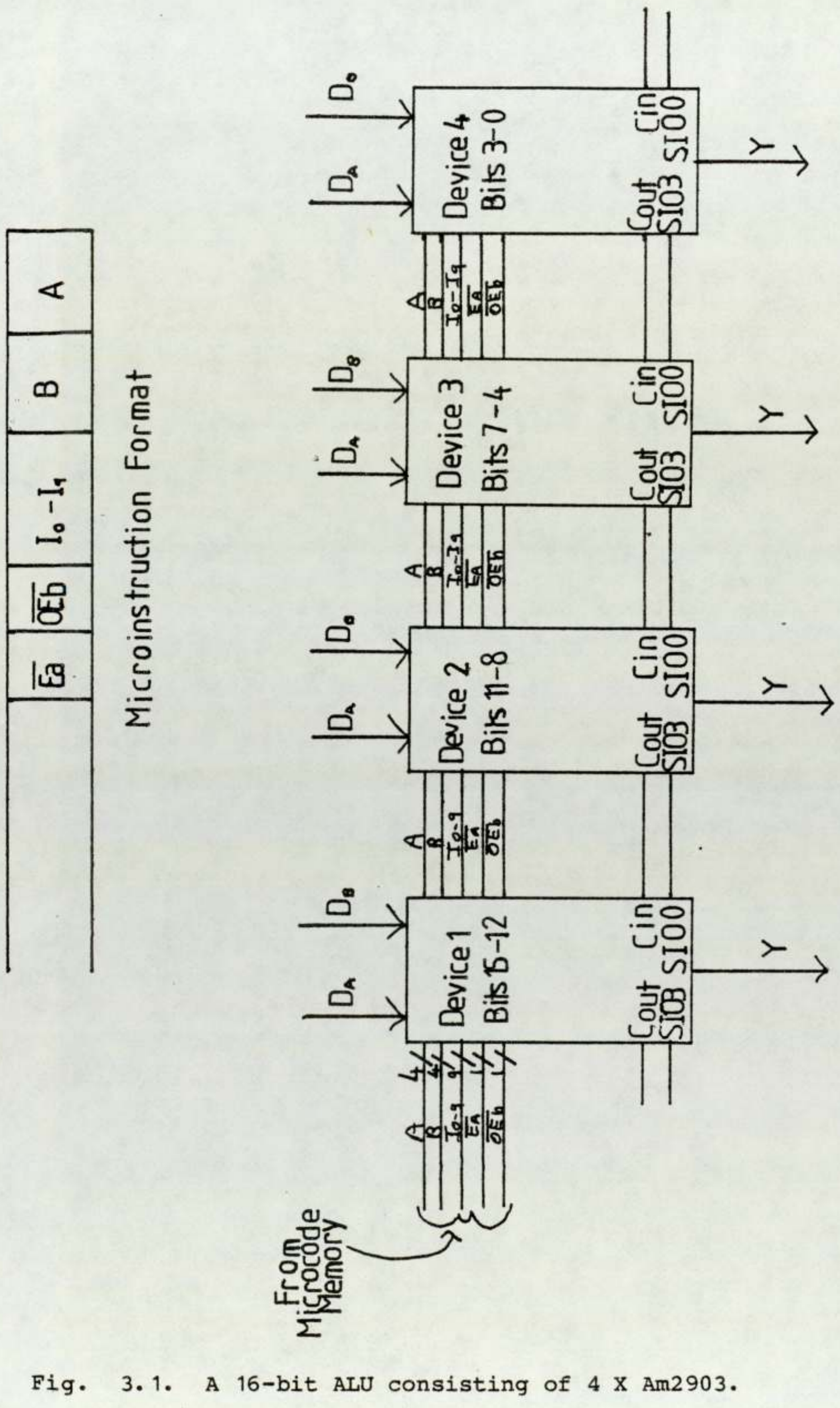


Fig. 3.1. A 16-bit ALU consisting of 4 X Am2903.

$\overline{Ea}$	I0	$\overline{OEb}$	ALU operand R	ALU Operand S
L	L	L	A-port Register	B-port Register
L	L	H	A-port register	DB
L	H	X	A-port Register	Q Register
H	L	L	DA	B-port Register
H	L	H	DA	DB
H	H	X	DA	Q Register

Note

1) L = LOW, H = HIGH, X = DON'T CARE.

2) The Q Register is an extra internal scratch register independent of the 16 RAM registers. It can be written to or read from.

Table 3.1. Am2903 Operand Source Control.

I4	I3	I2	I1	I0	ALU Functions
L	L	L	L	L	Special Functions
L	L	L	L	H	Fi = HIGH
L	L	L	H	X	F = S - R - 1 + Cin
L	L	H	L	X	F = R - S - 1 + Cin
L	L	H	H	X	F = R + S + Cin
L	H	L	L	X	F = S + Cin
L	H	L	H	X	F = $\overline{S}$ + Cin
L	H	H	L	L	Reserved
L	H	H	L	H	F = R + Cin
L	H	H	H	L	Reserved
L	H	H	H	H	F = $\overline{R}$ + Cin
H	L	L	L	L	Reserved
H	L	L	L	H	Fi = LOW
H	L	L	H	X	Fi = Ri AND Si
H	L	H	L	X	Fi = Ri EXCLUSIVE NOR Si
H	L	H	H	X	Fi = Ri EXCLUSIVE OR Si
H	H	L	L	X	Fi = Ri AND Si
H	H	L	H	X	Fi = Ri NOR Si
H	H	H	L	X	Fi = Ri NAND Si
H	H	H	H	X	Fi = Ri OR Si

Note

1) L = LOW, H = HIGH, X = DON'T CARE.

2) i = 0 to 3.

3) ALU Special Function details have been ommitted for the sake of simplicity.

Table. 3.2. Am2903 Function Control

I8	I7	I6	I5	Destination
L	L	L	L	Arithmetic Shift Right, store result in B-port Register
L	L	L	H	Logical Shift Right, store result in B-port Re gister
L	L	H	L	Arithmetic Shift Right, store result in B-port Register *
L	L	H	H	Logical Shift Right, store result in B-port Re gister *
L	H	L	L	No Shift, store result in B-port Register
L	H	L	H	No Shift, result goes to YBUS only *
L	H	H	L	No Shift, store result in Q Register
L	H	H	H	No Shift, store result in B-port and Q Registe rs
H	L	L	L	Arithmetic Shift Left, store result in B-port Register
H	L	L	H	Logical Shift Left, store result in B-port Reg ister
H	L	H	L	Arithmetic Shift Left, store result in B-port register *
H	L	H	H	Logical Shift Left, store result in B-port reg ister *
H	H	L	L	No Shift, result goes to YBUS only
H	H	L	H	No Shift, result goes to YBUS only *
H	H	H	L	Sign extend, result goes to B-port Register
H	H	H	H	No Shift, result goes to B-port Register

Note

1) L = LOW, H = HIGH, X = DON'T CARE

2) Destinations marked '*' also shift the Q Register independently of the main ALU results. Details have been ommitted for simplicity.

3) Details of the sign extend facility and parity generation have also been ommitted.

4) All results appear on the YBUS regardless of I8 - I5

Table 3.3. Am2903 Destination Control

signals directly connected from the current microinstruction register to the control inputs of the various devices within the processor.

The ALU is responsible for all the various arithmetic operations required. These fall into three categories:-

1) Those that are implicit to a certain machine-code instruction (e.g. "ADD", "SUB" etc.).

2) The generation of addresses for memory read and write operations. This would include incrementing the program counter before fetching the next machine code instruction. Also, incrementing and decrementing the stack pointer.

3) The generation of any addresses explicit in a machine code instruction.

The ALU receives control signals from the CCU such as the operation required (Add, Subtract etc.) and the data sources (memory, internal registers etc.). In return it sends back the result of any operations (>0, <0, =0 etc.). This enables the CCU to make conditional jumps based on these results.

Finally, the main memory is used to store the computer program, consisting of machine code instructions, together with any data that is required. It receives control signals from the CCU (memory request, read/write etc.).

At the start of a power-on sequence the CCU instructs the ALU to generate the address for the first machine-code instruction. After this has been done a memory-read request is made. The first instruction will then be read from memory. After this, a Decoder in the CCU will translate its op-code into a microcode memory address. This will correspond to the location where the microcode to execute the current machine instruction is stored. After its execution has taken place, the microcode sequencer will jump to the the fetch cycle microcode. Typically, this would consist of incrementing the program counter (a register in the ALU) or generating a jump address, requesting a memory read and decoding the op-code as before. The next instruction would then be executed and control would continue in this manner.

3.1.3 An Advanced Bit-Slice Architecture

In order to produce a faster and more efficient processor the technique of machine-code pipelining would be adopted in a bit-slice architecture. This is achieved by adding two extra functional sub-systems to the computer; a Program Control Unit (PCU) and a Datapath (Machine Instruction Pipeline) as shown in Fig. 3.3.

The philosophy behind speeding up the processor is quite simple. There are two Arithmetic Units, one generates the program counter addresses and the other one performs the arithmetic operations required by the machine code. This means that there is no need to wait until the current instruction is executed before the next one is fetched. It can be done simultaneously. This notion can be taken even further since there are two steps involved in a fetch cycle; forming the address and then reading the instruction. A pipelined machine will be performing three functions at any one time. It will be executing the current instruction, reading the

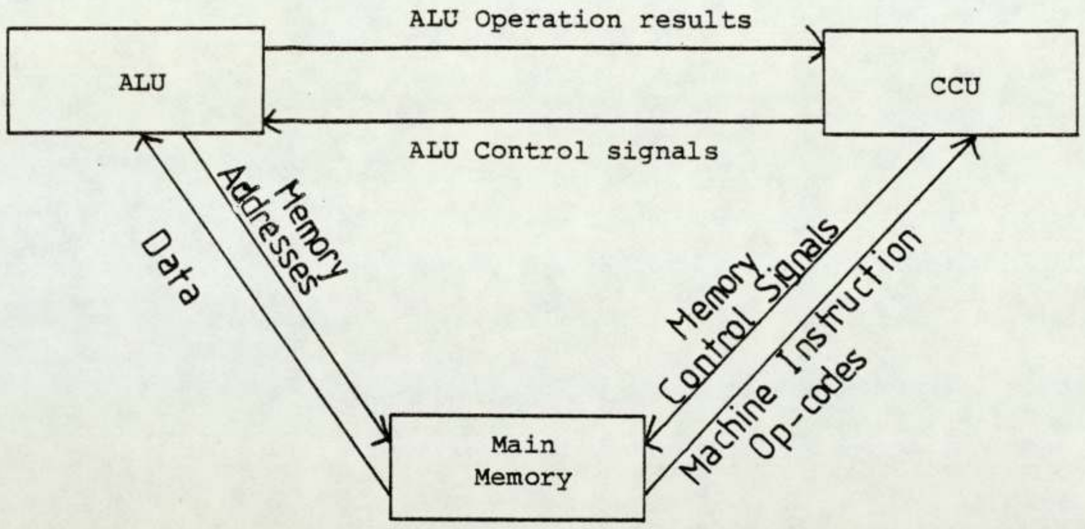


Fig. 3.2. A Simple Bit-Slice Computer.

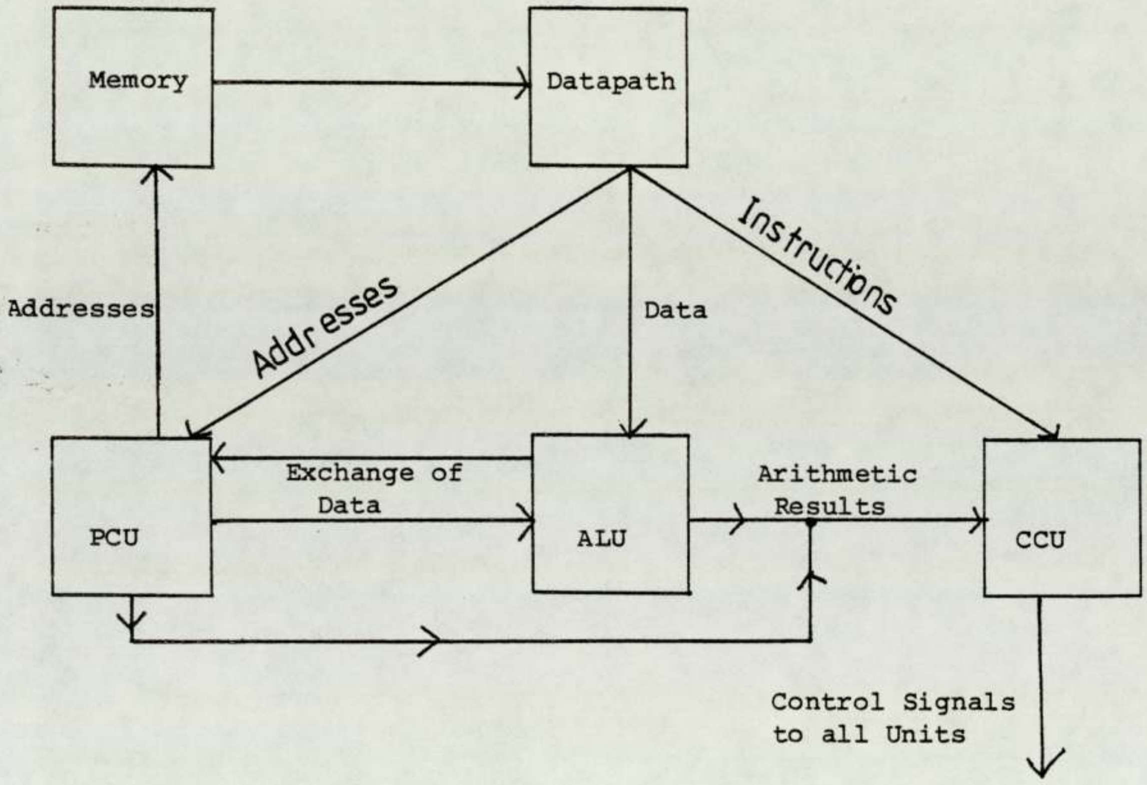


Fig. 3.3. An Advanced Bit-Slice Architecture.

next one and generating the address for the succeeding instruction.

This process clearly speeds up the processor as long as the machine code is reasonably sequential. If there are many jumps at machine-code level then the advantage tends to be lost. This is because the processor is not aware of the jump until the instruction is executed. By this time the next instruction has been fetched and the address for its successor has been generated. Both of these must be discarded. However, a pipelined processor would still be as fast as a non-pipelined machine even if every instruction was a jump.

Because concurrency exists in the processor design another sub-system is required. This would be where data, which has been read from memory and is waiting to be processed, can be stored. To do this the Datapath is introduced. This is merely a series of registers in which values read from memory can be "pipelined" en route to one of the ALUs or the CCU.

3.2 The Super Sixteen Processor

3.2.1 Origin of the Super Sixteen

The Super Sixteen processor was obtained from [3.2]. It had been originally designed to demonstrate the principles of a microprogrammed system. In particular, it used appropriate techniques to maximise the processor's throughput. Accordingly, it uses two arithmetic units; the main ALU and the Program Control

Unit (PCU) which generates memory addresses.

The Super Sixteen was chosen for three main reasons. The first was its fast memory accesses. There is no time overhead involved in memory operations provided that the address is generated two microcycles before the data is required. This makes the Super Sixteen particularly suitable for executing P-code with its large number of memory accesses, particularly to the stack.

The second reason for the choice was its addressing modes. P-code required 8-bit byte or 16-bit word addressing. All words are accessed on an even boundary as shown in Fig. 3.4. The Super Sixteen is capable of this.

The final reason was that machine-code pipelining and the inherent requirement of an extra ALU introduced an element of redundancy which could be used to achieve fault-tolerance. This concept is discussed fully in the next Chapter.

Certain modifications to the original design were made, these were due to the following reasons :-

1) There were errors in the original design.

2) The Datapath was configured for a 2-addressing instruction set. P-code is a zero-addressing machine code.

3) Significant hardware difficulties were experienced with the original memory board. This was replaced with a standard equivalent used within the department. The new board included memory-mapped

I/O. It was therefore considered practical to alter the I/O.

4) The interrupt hardware was simplified by the introduction of polling. This meant that the implementation of I/O was simpler. A slight loss in processor speed might be obtained but this would not affect the demonstration of fault-tolerance principles. Also, a timer interrupt was introduced. This was necessary for performing real-time scheduling which is a basic requirement of Concurrent Pascal.

5) The CCU required test results from the PCU as well as the ALU.

3.2.2 Hardware Description

3.2.2.1 The Processor

The processor organisation is shown in Fig. 3.5, it consists of five functional sub-systems; the program control Unit (PCU), the Arithmetic and Logic Unit (ALU), the Computer Control Unit (CCU), the Datapath and the Timer Interrupt Unit (TIU).

The PCU is responsible for generating memory addresses. Normally, it only needs to push and pop the stack and increment the program counter. However, occasionally, it must reload the program counter (for jumps) and reposition the stack pointer (for example, when exiting subroutines). The PCU, therefore, only needs to be able to perform addition and subtraction operations. The Program Counter (PC) and the Stack Pointer (SP) are represented by two internal registers. To make the PCU more efficient three of the internal registers permanently hold the numbers 1,2 and 4. This allows an increment or decrement by these values to be made internally to the PCU. Table 3.4 shows the detailed register assignment.

The PCU is built from 4 X Am2901s in a similar manner to that described in 3.1.1. It produces a Z-output which is '1' if the result of an arithmetic operation is zero and '0' otherwise. This boolean output is routed to the CCU via a Test Tree circuit, which enables it to act on the results of conditional tests made within the PCU.

The PCU can use either an internal register as a source or the DA bus. This allows it to receive immediate data from the CCU or values read from memory. Its output can be loaded into the Memory Address Register (MAR) or transferred to the YBUS via the PCU Transceiver (PCUTRAN).

The ALU sub-system is constructed from 4 X Am2903 devices and an Am2904 Status and Shift Control Unit. The Am2903 has better arithmetic facilities than the Am2901. For example, these speed up multiplication and division. It has three external buses; the DA bus, the DB bus (which is not used) and the YBUS. Both the DB bus and the YBUS can be tri-stated and used as external inputs or outputs. Immediate values from the CCU or data read from memory can be input along the DA bus. Data output onto the YBUS can be latched into the Data Register (DREG) for writing into memory or the Transfer Register (TREG) for transferring to the DA bus or to the MAR via the PCUTRAN. If the Y outputs on the Am2903 are not enabled

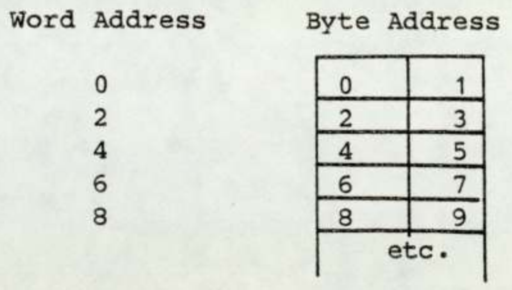


Fig. 3.4. P-code Addressing Requirement.

Register No.	Assignment
0	PC
1	SP
2	1
3	Current Concurrent Pascal Process Head Address
4	2
5	4
6	Scratch Register
7	Scratch Register
8-15	Not Used (Wired Disable)

Table 3.4. PCU Register Assignment.

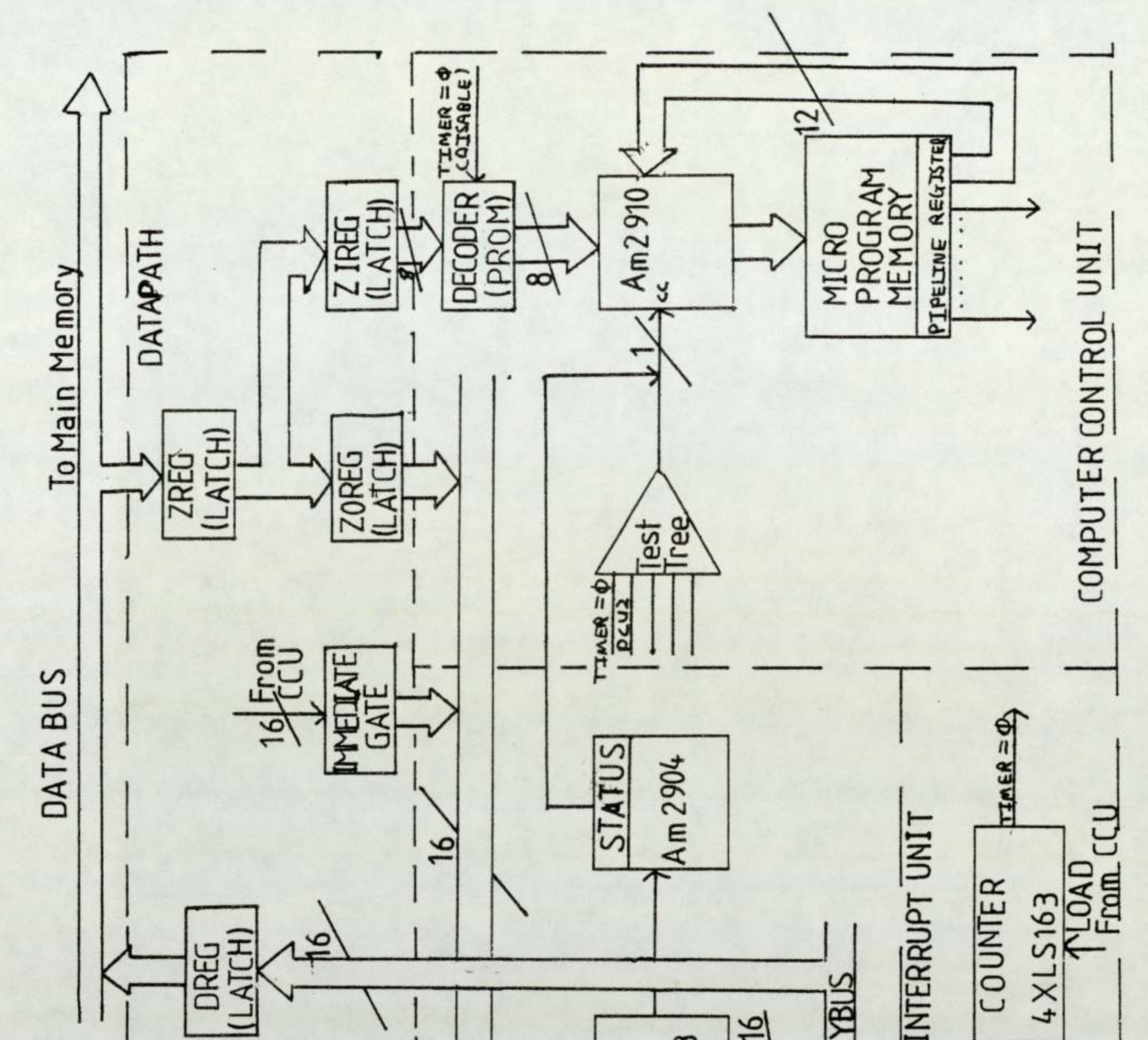


Fig. 3.5. Block Diagram of the Super Sixteen Processor.

then they will become inputs. This allows data transferred from the PCU via the PCUTRAN to be loaded into the ALU. The TREG can be used to transfer data from the ALU to the PCU.

There are four operation result outputs produced by the ALU; the Z, N, C and O bits. The Z-bit indicates whether the ALU result is zero, the N-bit is the sign of the result and the C and O bits respectively represent Carry-out and Overflow. These bits are transferred to the Am2904. Here they are converted into boolean values (>, >=, <, <=, =, NOT =) for signed and unsigned comparisons for the test required by the Am2904 instruction signals. These inputs are connected directly to the CCU. As well as being a Test Status Multiplexer the Am2904 provides Shift Linkage and Carry-in values for the ALU.

The 16-bit Datapath is mainly constructed from 2 X 8-bit devices, most of which are latches. Any data read from memory is received through the Z register (ZREG). This data can then be routed onto the DA bus via the Z0 register (Z0REG) or to the CCU Decoder via the ZI Register (ZIREG). From the DA bus, data can then be routed to the ALU or the PCU. If the data is transferred to the CCU Decoder then it is treated as an op-code. The Decoder is a PROM whose address inputs are connected to the ZIREG and whose 8-bit data outputs are connected to the Am2910 microcode sequencer. The Decoders function is to convert an op-code into a microcode address where it will be executed.

The CCU controls the order of the execution of microinstructions. The Am2910 microprogram controller sequences the microcode. It contains an internal incrementer. When no jump is made the next microinstruction is fetched from microprogram memory by incrementing the microprogram counter. In order to be able to perform jumps the Am2910 can receive branch addresses from the current microinstruction register. In order to be able to execute machine code instructions the Am2910 is also connected to the mapping PROM. When a "jump to map" instruction is executed the Am2910 loads up the microprogram counter with the current mapping PROM value. This causes a jump to the microcode to execute that instruction. The Am2910 also contains an internal counter for performing fixed duration loops independently of any external test results. The microcode memory is 96 bits wide and consists of 12 X 8-bit EPROMs. The current microinstruction register, also known as the pipeline register, consists of 12 X 8-bit latches.

The Timer Interrupt Unit (TIU) is a 16-bit counter consisting of 4 X 4-bit sub-units. Whenever the counter reaches zero it causes an interrupt. The interrupt bit disables the mapping PROM Decoder. This means that the next time a "jump to map" instruction is executed by the Am2910 it will cause the Sequencer to jump to a wired-in interrupt address. An interrupt will only occur at the end of the execution of a machine code instruction since this is the only time that a "jump to map" instruction is executed. At the start of the microcode interrupt sequence the TIU counter will be reloaded with a value from the pipeline register corresponding to the interrupt period. This causes the next interrupt to be generated when the counter reaches zero again. The interrupt signal is also routed to the Test Tree. This allows the CCU to detect interrupts (by polling) when machine code is not being executed.

3.2.2.2 Memory and Input/Output

The author experienced hardware difficulties with the original Super Sixteen memory design. It was decided that the best action was to replace it with a memory board based on a standard piece of hardware used within the department. The original memory board had a complicated timing and handshake mechanism. Much of this is no longer used. However, since the timing signals are all inter-dependent they still appear in the circuit.

The memory unit eventually used was based on a Motorola M6809 processor board which was used for microprocessor work within the department [3.3]. The processor was removed and the board was used as a normal memory. The M6809 board was only 8-bits wide and so two such units had to be cascaded together to provide a 16-bit memory. An extra board was built to select one or both of these depending on whether a byte or word memory request was made (Fig. 3.6).

The memory board uses memory-mapping of I/O. This means that each I/O device has a specific memory address. To access it the processor merely performs a read or write to the correct location. This makes I/O requests quite simple, they are merely a sequence of memory accesses. All I/O devices are 8-bits wide and appear on the least significant of the two memory boards.

The memory contains 8K of RAM and 16K of EPROM in which Concurrent and Sequential Pascal programs can be stored. It also posseses an Asynchronous Communications Interface Adapter (ACIA) which can be used to access a terminal and up to five Peripheral Interface Adapters (PIA) to communicate with other processors. Only

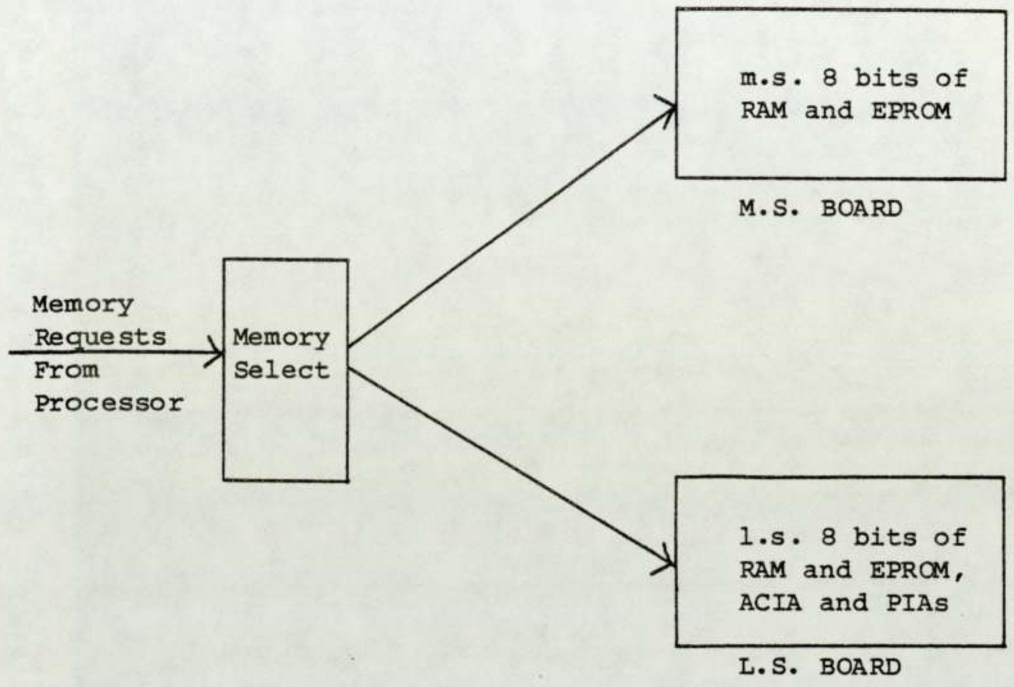


Fig. 3.6. Constructing a 16-bit memory with word and byte addressing from 2 X 8-bit memory boards.

one of these PIAs is needed for a dual processor system. The logic diagrams for the memory board appear in Appendix 6.

3.2.3 Control of the Processor

All parts of the Super Sixteen, including the CCU itself, are under microcode control. Normally, the CCU is executing machine code (P-code), there are four basic stages involved in this :-

1) Form the instruction memory address.
2) Fetch (read) the instruction.
3) Decode the op-code.
4) Execute the instruction.

As an example take a simple P-code instruction with no operands. This would normally read two values off the stack, perform an arithmetic operation on them (add say) and then write the result back onto the stack. This could be implemented as shown in Table 3.5.

The structure of the Super Sixteen, however, makes several of these operations concurrent. Also, the first three stages of the next P-code instruction (Form address, Fetch, Decode) can be executed simultaneously. This means that the control cycle would be as shown in Table 3.6. Several points should be noted from this :-

1) More than 3 microcycles are saved as Tables 3.5 and 3.6 suggest. This is because microcycles 1 to 3 would be executed concurrently with instruction n-1. Therefore, with an instruction such as this,

Microcycle	Operation	Comment
1	PC -> MAR	Form Address
2	Memory Read Request	Fetch Instruction
3	ZREG -> ZIREG and Jump to Map	Decode
4	SP -> MAR	Start of Execution :- Generate Address for First Operand
5	Memory Read Request	Fetch First Operand
6	ZREG -> ALU reg. x	Store First Operand in ALU register number x
7	SP + 2 -> MAR	Generate Address for Second Operand
8	Memory Read Request	Fetch Second Operand
9	ZREG + ALU reg.x -> DREG	Execute Instruction
10	SP -> MAR	Generate Stack Address
11	Memory Write Request	Put Result on Stack

Table 3.5. Non-pipelined execution of a typical P-code instruction.

Microcycle	Operation	Comment
1	PC -> MAR	Form Address for instruction n
2	Memory Read Request, PC + 2 -> MAR	Form Address for instruction n+1 and fetch instruction n
3	Memory Read Request PC + 2 -> MAR Jump to Map	Fetch Instruction n+1, Form Address for Instruction n+2 and Decode Instruction n
4	SP -> MAR, ZREG -> ZIREG	Generate Address for First Operand and move Instruction n+1 down the Pipeline
5	Memory Read Request, SP + 2 -> MAR	Fetch First Operand and Generate Address for Second Operand
6	ZREG -> ALU register x, Memory Read Request, PC -> MAR	Store First Operand in ALU register x, Fetch Second Operand and Form Address for Instruction n+2
7	ZREG + ALU reg. x -> DREG Memory Read Request, SP -> MAR	Execute Instruction, Fetch Instruction n+2 and Generate Address and Data for Writing Result to Stack
8	Memory Write Request, PC + 2 -> MAR, Jump to Map	Write Result to Stack, Generate Address for Instruction n+3 and Decode Instruction n+1

Table 3.6. Pipelined Execution of a typical P-code Instruction.

pipelining effectively halves the execution time.

2) The address for instruction n+2 is generated twice, once at microcycle 3 and again at microcycle 6. It would, in this case, save an extra cycle if the stack address was instead generated at microcycle 3. However, at least half of the P-code instructions have several operands following them in memory. These require the program counter address to be in the MAR following the decode operation. All P-code instructions are independently executed and can appear in any order. It is therefore necessary to standardise the pipeline state at the start of the execution of each instruction. It was decided to arrange the microcode so that all P-code instructions could assume that the next memory word was in the ZREG and the address of the word after that was in the MAR. Each P-code instruction is responsible for leaving the processor in the same state for its successor.

If there is a timer interrupt then the P-code will be interrupted when the next decode (Jump to Map) is performed. When the CCU has finished serving the interrupt it must refill the pipeline in a similar manner to that shown in microcycles 1 to 3 of Table 3.6.

In the event of a fault in some other functional unit being detected the CCU can continue to operate. However, no P-code will be executed nor any decodes performed (unless testing the Decoder).

The generation of both P-code and microcode and their testing requires quite a complex development system. A description of this can be found in Appendices 1 and 4. Full listings of the microcode, together with documentation appears in Appendices 9 and 7 respectively.

Chapter 4

Fault-Tolerance Theory

4.1 A Multiprocessor Architecture

The basic philosophy of using a microprogrammable machine, such as the Super Sixteen, to achieve fault-tolerance is that the CCU sub-system is the only functional unit whose failure would prove catastrophic. However, it is no use having an operational CCU unless it has some computational facilities such as an arithmetic capability. The architecture of the Super Sixteen is such that these computational facilities will always be available to the CCU provided that not more than one functional unit fails since it has access to two arithmetic units, the ALU and the PCU. The essential requirement of the theory to be described is that one of these units will be operational, even though the CCU is unaware of which one. There is, therefore, a basic duplication of components in a processor architecture such as this to achieve high performance. However, in the event of a failure this can be regarded as a form of redundancy to achieve fault-tolerance (Fig. 4.1).

In this chapter a method will be described whereby the CCU will attempt to diagnose which functional unit has failed. To achieve this the CCU must test each unit within the processor architecture, this includes the main memory and the Datapath as well as both ALUs. One or both of the ALUs must be used to carry out these tests. This poses a dilemma. Until the tests have been evaluated it is not clear which of the two ALUs is at fault. On the other hand the tests cannot be analysed until it is known which ALU is operational.

Therefore, it would seem that a faulty processor may have the ability to perform a set of tests at microcode level but it does not have the ability to evaluate correctly these tests in order to make a fault diagnosis. Consequently, a reliable means of analysing these results must be defined.

Consider the computer system as a whole. Operating in real time it will have a set of tasks to perform. The control of these tasks is in the form of machine code (P-code) stored in main memory. It may be that the main memory has failed or that the address generation mechanism to access the main memory is at fault in which case the machine code cannot be accessed. It would be unreasonable to expect the microcode to assume all of these tasks on an already degraded processor, in any case this would defeat the entire purpose of having machine code and high level languages. There must be some type of multiprocessor architecture whereby one or more processors can assume the tasks of the faulty processor. In order to simplify the multiprocessor architecture only one extra processor will be used. In addition to taking over the tasks previously performed by the faulty processor, the second processor could also perform the function of evaluating the microcode test results.

Hence, the solution proposed is to perform a set of tests at microcode level within the faulty processor. Their results can then be passed to another processor which will evaluate them at HLL level and, hopefully, diagnose the fault. It would be useful at this stage to define some terms of reference (Fig. 4.2).

A <u>fault</u> is defined as a failure of a component within a functional subsystem which causes it to operate incorrectly. A

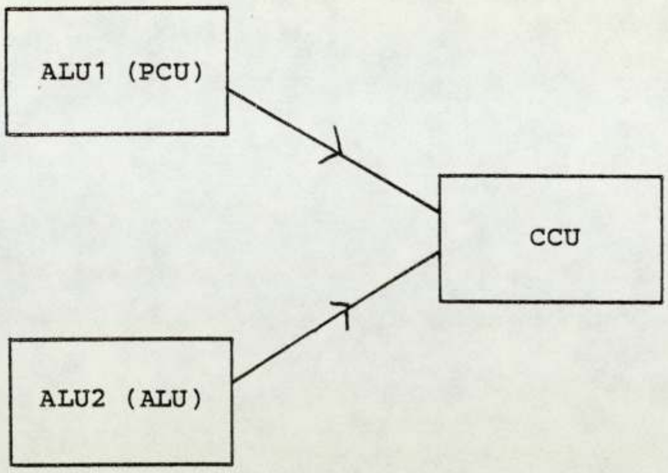


Fig. 4.1. The Basic Redundancy of a High Speed Bit-slice Computer.

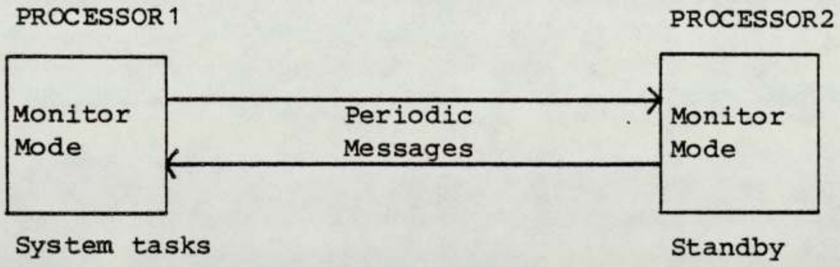


Fig. 4.2 a). Processor Status - Both processors non-faulty.

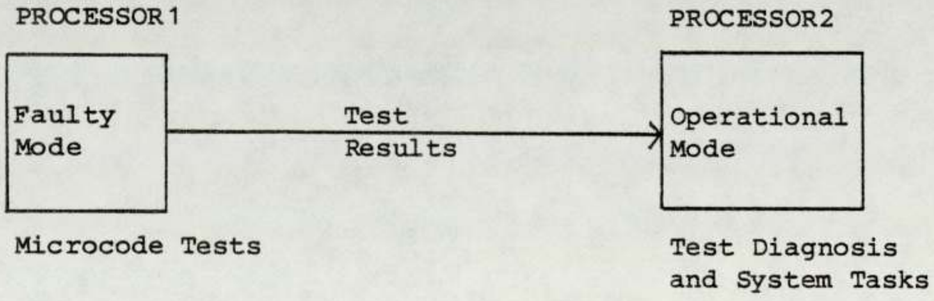


Fig. 4.2 b). Processor Status - One processor Faulty.

single fault will cause only one sub-system to fail. A multiple fault will prevent several functional units from operating as required.

The Faulty (or failed) Processor is the processing unit which has a fault and is attempting to perform a set of microcode tests.

The Operational Processor is that which is error-free and has assumed all the tasks performed by the Faulty Processor and is also evaluating the test results generated by it.

The Main Processor is the processor which performs all the real-time tasks required of the system. In an error-free two-processor system this could be either processor, if there is a fault present in the system then clearly it will be the Operational Processor. This corresponds to the traditional main/backup real-time fault-tolerance approach.

The Backup Processor is on standby should a fault occur. If the Main Processor does fail then it will become the Main Processor and assume all the system tasks.

If the system is error-free then both processors will be in monitor mode. This is completely independent of the main/backup status of the processor. A processor in monitor mode listens for periodic messages from its neighbouring processor in order to make sure that it has not failed. If a fault occurs in the backup processor then this will be diagnosed in the same way as a failure in the main processor, the only difference being that transfer of task responsibility does not take place.

4.2 Microcode Tests - The Faulty Processor

This section describes a set of microcode tests the results of which will uniquely determine certain fault conditions within the processor. These tests will apply to single failures and it is assumed that multiple faults are not present.

4.2.1 The Arithmetic Units

Although the Super Sixteen uses the PCU to generate memory addresses and the ALU to produce memory data it is often useful in practice to be able to exchange these roles. For this reason, the Super Sixteen incorporates the PCUTRAN which allows the transfer of ALU data to the MAR and PCU values to the DREG. In addition the PCUTRAN, together with the TREG, can be used to convey data between the two arithmetic units. Fig. 4.3 shows this section of the Super Sixteen.

The facility of being able to use either Arithmetic Unit for any purpose and the capability to transfer data between them is very useful from the point of view of fault-tolerance since they can be used to test each other. Also, memory tests can be performed with all permutations of two out of the three functional units (ALU, PCU and PCUTRAN). If there is only one fault present then the only test that will pass will be the one that does not use the failed unit. This means that it will be a straightforward matter to diagnose the fault. If the main memory, which is being tested, is faulty then this can also be detected since all three tests will fail and no other single functional unit failure would cause all of the memory

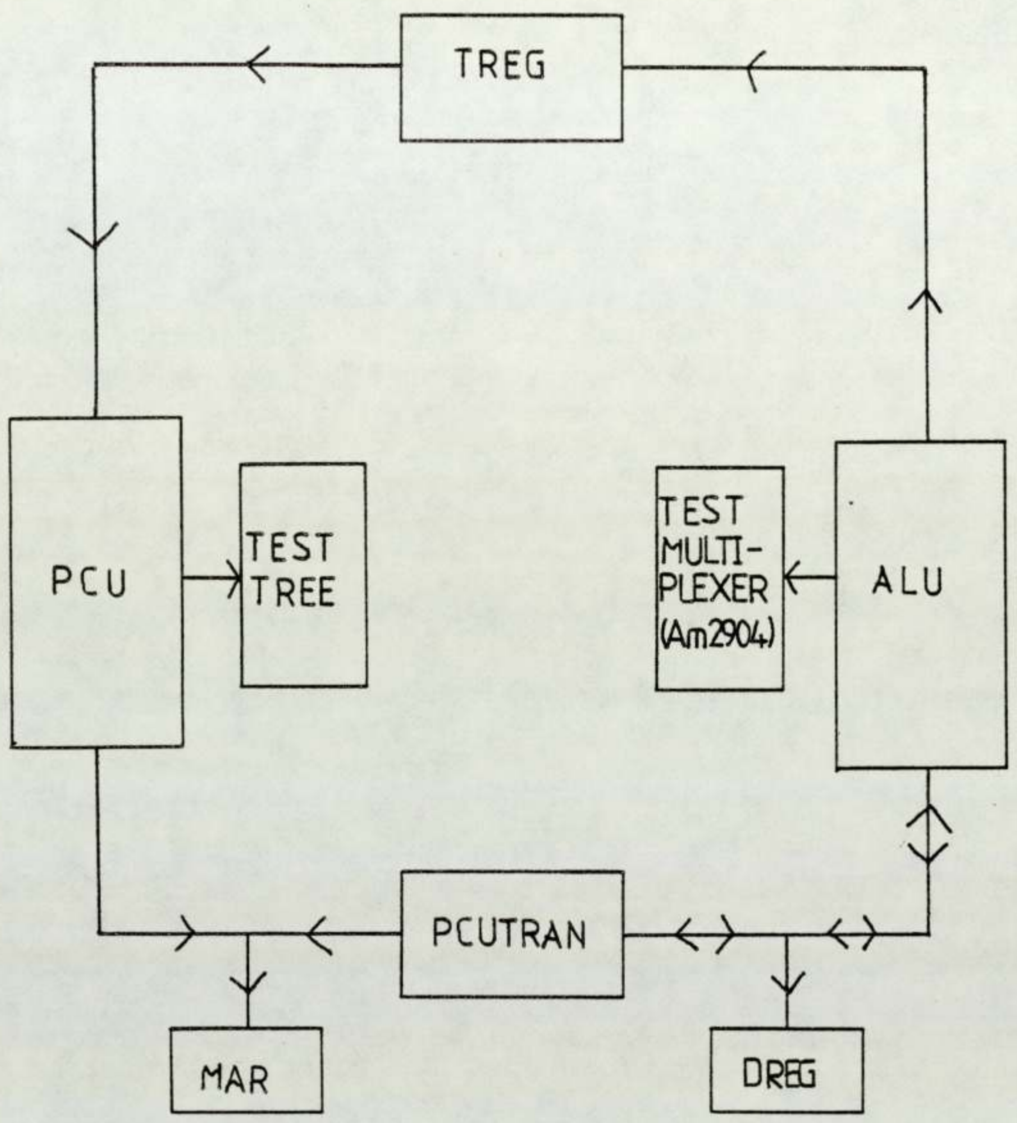


Fig. 4.3. Transfer of data between the ALU and the PCU.

tests not to pass.

To clarify the situation the three tests are as defined below :-

<u>Test A</u>

Without using the PCUTRAN, test the memory using the PCU to generate addresses and the ALU to produce and evaluate other data.

<u>Test B</u>

Without using the PCU, test the memory using the ALU and the PCUTRAN to generate addresses and the ALU to produce and evaluate other data.

<u>Test C</u>

Without using the ALU, test the memory using the PCU to generate addresses and evaluate data and the PCU and the PCUTRAN to produce data.

If the PCUTRAN is faulty then test A is the only one that will pass since it is the only test that does not use that unit. Similarly, only test B will pass in the event of a PCU failure and only test C will pass if the ALU is faulty. If all three tests fail then the PCU, the ALU and the PCUTRAN are all in agreement that the memory is faulty. If a notation is adopted whereby a boolean variable indicates whether a test has passed or not then their results can be shown using an appropriate expression. For example, if :-

'A' indicates that test A has passed

and '$\overline{A}$' indicates that test A has failed

then $\overline{A}.\overline{B}.\overline{C}$ means that tests A,B and C have all failed.

Having defined a set of tests and a notation to represent their outcomes the argument stated above can be re-iterated in a concise manner :-

$\overline{A}.\overline{B}.\overline{C}$ Indicates a memory failure.

$A.\overline{B}.\overline{C}$ Indicates a faulty PCUTRAN.

$\overline{A}.B.\overline{C}$ Indicates a faulty PCU.

$\overline{A}.\overline{B}.C$ Indicates a faulty ALU.

Since none of the three tests use the TREG its failure will not be detected. It is therefore necessary to introduce a further test which will detect this condition.

<u>Test D</u>

Pass some data from the ALU to the TREG, through the PCU and the PCUTRAN and back to the ALU. Use the ALU to see if the value is the same after its passage through the machine.

Although the first three tests defined above will diagnose any fault within the ALU or the PCU they will not identify the specific component within that functional unit that has failed. For example, if the ALU is diagnosed as faulty then this might be due to the actual arithmetic unit (4 X Am2903s) being faulty or it could be because the Test Status Multiplexer (Am2904) has failed. Similarly, if a fault is diagnosed within the PCU it might be the Am2901 devices that have failed or it could be due to a faulty Test Tree.

Test D combined with the other three tests will diagnose the fault within the PCU. This is because a faulty PCU will cause test D to fail as well as tests A and C since it requires the PCU to transmit the value through it. However, if the Test Tree is faulty then test D will pass since it does not use this unit (Test A will also pass).

However, to diagnose a faulty Test Status Multiplexer within the ALU another test must be defined. This test, like the PCU in test D, requires the ALU to pass a value without making any comparisons on it. This raises the question of where the value should be checked. Since the purpose of the test is to avoid using the Am2904 Test Status Multiplexer it cannot assume this responsibility. However, the Faulty Processor already transmits the results of all tests to the Operational Processor (in the form of message codes). There is no reason why the actual value used in the test could not be transmitted to the Operational Processor which would then decide whether it was correct or not. The test, therefore, is defined as follows :-

<u>Test E</u>

Pass some data through the ALU and across to the Operational Processor. The Operational Processor will examine this value to check whether it has been corrupted.

A faulty Test Status Multiplexer will be detected by the outcome:-

$\overline{A}.\overline{B}.C.\overline{D}.E$

since C and E are the only tests which do not use it.

4.2.2 The Datapath

With the set of five tests described in the previous section any fault in the ALU, the PCU or the transfer devices (PCUTRAN and TREG) can be uniquely determined. It has also been stated that a faulty memory can be diagnosed. This is a rather broad generalisation since a memory failure could be due to a faulty Datapath within the processor itself. A closer examination of the Datapath (Fig. 4.4) reveals that a more detailed diagnosis can be made.

Most of this sub-system is constructed from 2 X 8-bit devices cascaded together to form 16-bit registers. One possibility is that one of the m.s. devices in the Datapath has failed and not the memory. Since the Super Sixteen can perform byte memory addressing a fault in one of the m.s. components can be detected because accessing the memory in this mode will only require the use of the l.s. devices in the Datapath.

Therefore, another test must be defined :-

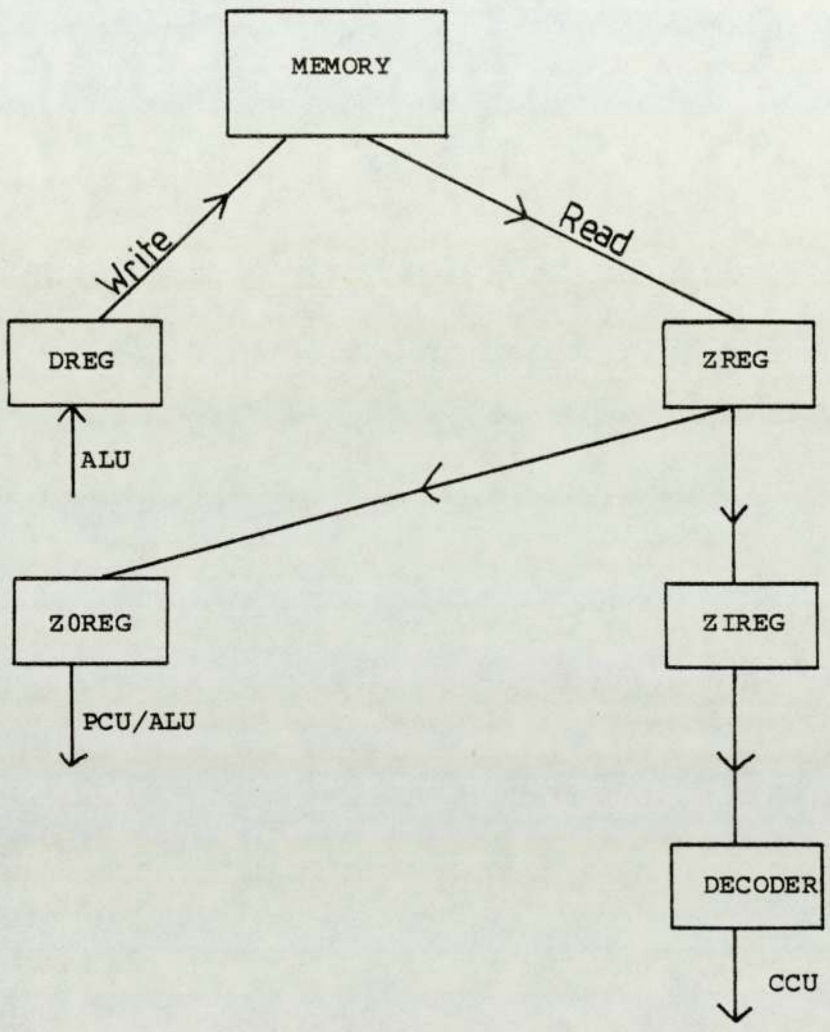


Fig. 4.4. Transfer of data through the Datapath.

Test F

Perform a memory test using byte addressing only.

If a faulty memory is detected and test F passes (denoted by $\overline{A}.\overline{B}.\overline{C}.D.E.F$) then one of the m.s. devices in the Datapath has failed.

However, there still remain two further possibilities :-

1) The ZREG is not faulty but the ZOREG has failed, thus causing all memory tests to fail.

2) Both the ZREG and the ZOREG are functioning correctly but the ZIREG or the Decoder is faulty. This would cause all memory tests to pass.

A test must be defined which checks the route from the ZREG to the CCU via the ZIREG and the Decoder. This test depends upon the co-operation of the Operational Processor. A machine-code instruction is defined, the function of which is to send a message to the Operational Processor. The test then has three stages :-

Test G

1) The Operational Processor sends the op-code of this machine instruction as a message to the Faulty Processor.

2) The Faulty Processor receives the message and routes it via

the ZREG to the ZIREG and the Decoder.

3) The Faulty Processor will attempt to execute the instruction, if successful it will send a message back to the Operational Processor.

If test G passes then it proves that the ZIREG and the Decoder are functioning correctly. This means that a faulty ZOREG will result in the test sequence :-

$\overline{A}.\overline{B}.\overline{C}.D.E.G$

If the m.s. part of the ZOREG has failed then test F (byte test) will pass but the test will fail if the l.s. half of the ZOREG is faulty. If the ZIREG or the Decoder is at fault then the test outcome will be :-

$A.B.C.D.E.F.\overline{G}$

A complete list of the tests that are performed are given in Table 4.1. A complete list of all the possible diagnosable faults, together with their test outcomes, is given in Table 4.2.

Test	Description
A	Test memory using the ALU to produce write-data and evaluate data read back from memory and using the PCU to generate memory addresses. The PCUTRAN is not used.
B	Test memory using the ALU to produce write-data and evaluate data read back from memory and using the ALU to generate memory addresses, loading the MAR via the PCUTRAN. The PCU is not used.
C	Test memory using the PCU to produce write-data and to evaluate data read back from the memory and using the PCU to generate memory addresses. The PCUTRAN is used to load the DREG. The ALU is not used.
D	Pass a value from the ALU to the TREG, through the PCU and the PCUTRAN and back to the ALU. Use the Am2904 Test Status Multiplexer to test whether the value has been corrupted.
E	Pass a value through the ALU and across to the Operational Processor which tests whether it has been corrupted or not.
F	Perform a memory test using byte addressing only.
G	The Operational Processor sends an op-code as a message to the Faulty Processor. If the message has been received then the Faulty Processor sends a message back.

Table 4.1. The set of microcode tests performed by the Faulty Processor.

Test Results	Diagnosis
$\overline{A}.B.\overline{C}.\overline{D}.E$	PCU Failure (Am2901)
$\overline{A}.\overline{B}.C.\overline{D}.\overline{E}$	ALU Failure (Am2903)
$A.\overline{B}.\overline{C}.\overline{D}.\overline{E}$	PCUTRAN Failure
$A.B.C.\overline{D}.E$	TREG Failure
$A.B.\overline{C}.D.E$	Test Tree Failure
$\overline{A}.\overline{B}.C.\overline{D}.E$	Test Multiplexer Failure (Am2904)
$\overline{A}.\overline{B}.\overline{C}.D.E.F$	ZREG (m.s.), DREG (m.s.) or Z0REG (m.s.) Failure
$\overline{A}.\overline{B}.\overline{C}.D.E.\overline{F}.\overline{G}$	Memory, MAR, ZREG (l.s.) or DREG (l.s.) Failure
$\overline{A}.\overline{B}.\overline{C}.D.E.F.G$	Z0REG (m.s.) Failure
$\overline{A}.\overline{B}.\overline{C}.D.E.\overline{F}.G$	Z0REG (l.s.) Failure
$A.B.C.D.E.F.\overline{G}$	ZIREG or Decoder Failure

Table 4.2. Test Results and their Diagnosis.

4.3 High Level Test Evaluation - the Operational Processor

Having defined a set of microcode tests to be performed by the Faulty Processor it is now necessary to define the functions to be performed by the Operational Processor using a High Level Language (Concurrent Pascal). This includes the real-time tasks required of the system when no faults are present.

In [2.9] Brinch Hansen uses example operating systems to describe Concurrent Pascal and how it can be applied. A similar approach is used here to describe the high level fault-tolerance operating system (FTOS) written for this project. The main aim of this section is to familiarise the reader with the high level concepts employed but it is hoped that a useful bi-product will also be to detail further the concepts of Concurrent Pascal.

4.3.1 Problem Specification

Since this project is concerned with the fault-tolerance aspect of a multiprocessor system the other functions required by the user are not especially important. However, whatever these functions are, one requirement will remain in any fault-tolerant system. There will be a database containing some information and it will be vital that its integrity be preserved. This will be done by using the classical approach of having one processor in main mode and its neighbouring processor in backup mode. A set of operations on this database will be defined which will only be executable on the main processor. Whenever an update is made a message will be sent to the backup system to perform the same task. Since, in this case, any user interface defined is merely for the purposes of demonstrating fault-tolerance it must be easy to change without altering the rest of the system. Fortunately Concurrent Pascal is very modular so this poses no difficulties.

The fault-tolerance aspect of the problem consists of updating (or receiving updates to) the database and of being able to detect and analyse a failed neighbour processor. The Backup processor must be able to assume control, thus becoming the Main Processor. When the user notifies the Main Processor it must re-accept a repaired standby unit.

The fault detection is achieved merely by the processors sending messages to each other in turn. If a message is not received within a certain time then it is deemed to have failed and the fault analysis will start.

All that is required in terms of fault analysis is to receieve the messages that are sent from the Failed Processor and, when the sequence is complete, to evaluate the results and output the diagnosis. The evaluation of the test results is merely a sequence of boolean comparisons. There is no need for the user to be aware of the fact that the processors are communicating. He only needs to receive information about the fault tolerance of the system should there be a failure. If this occurs he will be informed (by the Operational Processor) that there has been a failure and, if possible, the location of the fault.

The user functions chosen to be implemented were based on a simple database containing oil well details; their name, location and size. A set of operations were defined on the database :-

Well Status (W) Command

Display the database contents at the terminal.

Change (C) Command

Change an entry.

Delete (D) Command

Delete an entry.

Insert (I) Command

Insert a new entry.

In addition, two other operations were required :-

Processor Status (P) Command

Output the System status at the terminal i.e. is the processor in main or backup mode and is its neighbouring processor operational or faulty.

Re-instate (R) Command

Re-instate the neighbouring processor. This would be called on the Operational processor after the Faulty Processor had been repaired to indicate to the system that it should now be regarded as being functional again.

The Change, Delete, Insert and Re-instate commands are only available in main mode.

4.3.2 The Solution

The first step taken in achieving a solution to this problem was to analyse the tasks required, remembering that concurrency of functions was available. The tasks that need to be performed by the system are as follows :-

1) The User Functions.

2) Communication with the neighbour processor.

3) Detection and Analysis of any faults in the system.

It would therefore be logical to make each of these functions a separate Concurrent process. The next problem to be considered is how these processes need to communicate with each other :-

1) The USER process needs to be able to communicate with the fault detection and analysis process (call it DEBUG) to assertain whether the processor is in main or backup mode. It also needs to be able to communicate with the COMMUNICATOR process to update the Backup Processor when necessary. This

should only take place when the Backup Processor is not faulty.

Therefore, it seems reasonable to inform the DEBUG process when an update has been made and allow it to decide whether or not to inform the Backup Processor. Consequently, the USER process does not need to communicate directly with the COMMUNICATOR process.

2) The DEBUG process needs to communicate with the COMMUNICATOR process.

3) Both the USER and the DEBUG processes require access to a terminal.

The system interactions are displayed in Fig. 4.5 in the form of an access graph. It appears from this as though USER and DEBUG have access to individual terminals. This is also how it appears in the Concurrent Pascal program since both processes have their own copy of the TERMINAL class. However, before accessing it they call the RESOURCE monitor which delays the process if the terminal is already in use, thus ensuring that only one of them will use it at any one time.

The WELL UPDATES monitor contains the database and also some other essential system information such as the main/backup mode and the neighbouring processor faulty/operational status. The NEIGHBOUR monitor is used to transfer message requests. PROCCOMM is merely a set of routines for sending and receiving messages.

The final step in designing a solution to this problem is to

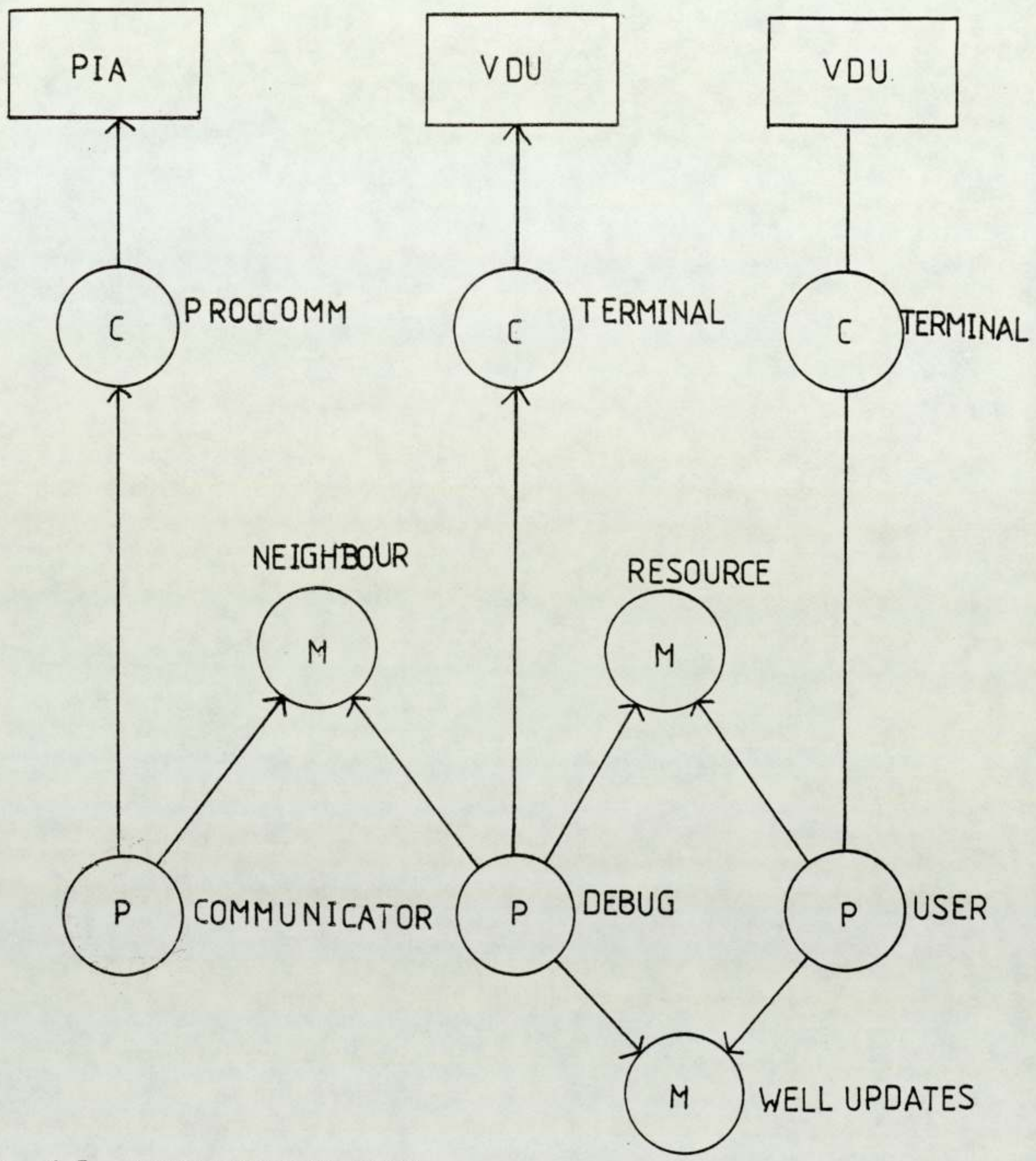


Fig. 4.5. Access Graph for FTOS.

decide which part of each process should be implemented in Concurrent Pascal and which should be implemented in Sequential Pascal (Fig. 4.6). Since the user functions should be easy to modify it is convenient to write them as a Sequential Pascal program. To alter them it would only be necessary to re-write this program. However, there must be a set of routines imbedded in the USER process in Concurrent Pascal to perform the following :-

1) Communicate with the terminal.

2) Read and write items from the database.

3) Read system information (main/backup mode etc.) and set the operational mode of the neighbouring processor in the event of the Re-instate command being invoked.

4) Inform the DEBUG process that an alteration to the database has been made.

These routines can be called from the Sequential Pascal program. In Concurrent Pascal terminology they are known as 'prefix procedures'.

The DEBUG process has two functions. If the neighbouring processor is operational then it has to detect a fault. Once this has been done it needs to perform the analysis. The process of fault detection merely consists of sending and receiving messages. This is a mutual operation since both processors are performing fault detection on each other. If a message is not received within a certain time then the neighbouring processor is assumed to be

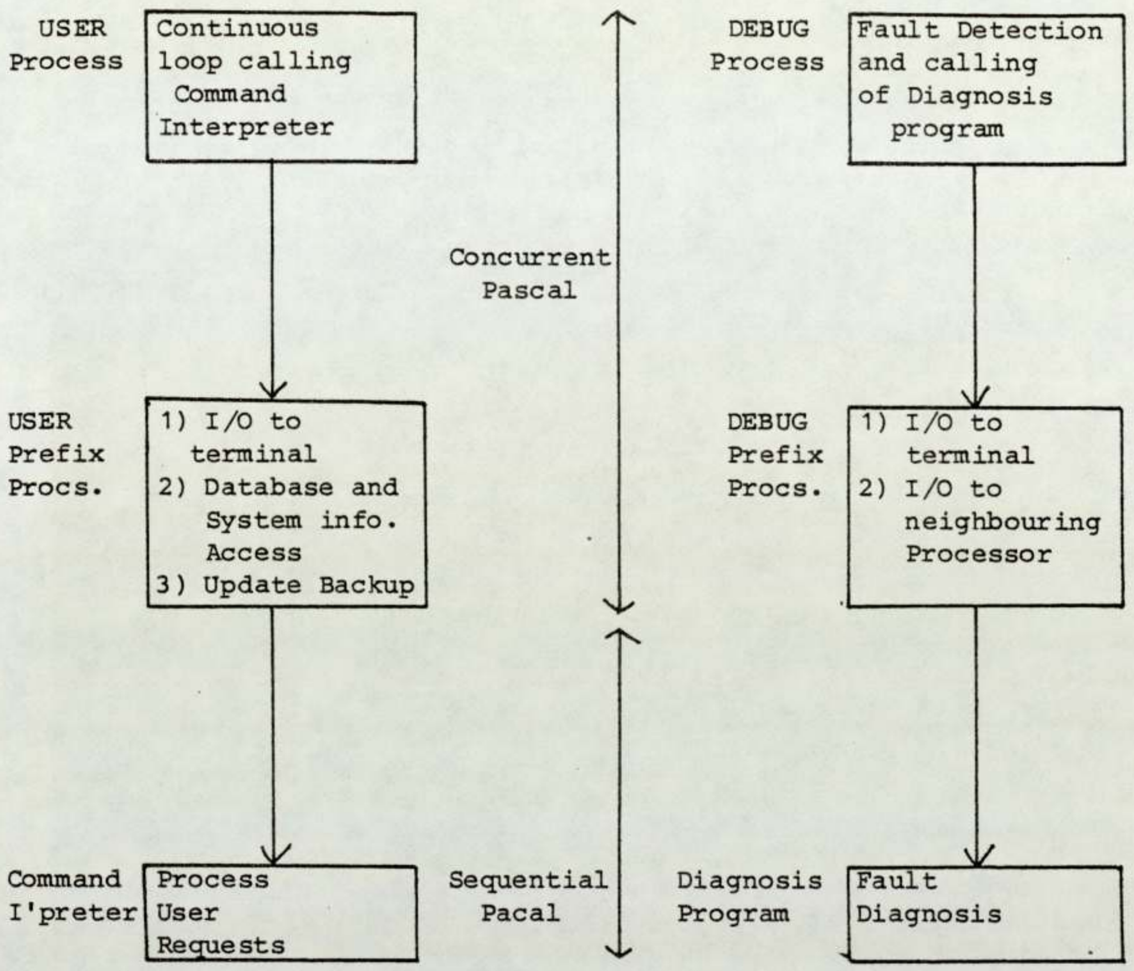


Fig. 4.6. Sequential/Concurrent allocation of FTOS functions

faulty and the analysis starts. The fault detection is fairly straightforward and does not merit a Sequential Pascal program. However, the fault analysis is more involved and could be altered as the fault-tolerance firmware evolves. Therefore, this should be written in the form of a Sequential Pascal program. This program should have access to prefix procedures to perform the following :-

1) Communicate with the terminal.

2) Send and receive messages (via the NEIGHBOUR monitor which acts as a postbox to the COMMUNICATOR process).

The Communicator process merely examines the NEIGHBOUR monitor for message requests and sends or receives them as required. If the neighbouring processor is faulty and a message is sent, but no reply is receieved, then COMMUNICATOR will be delayed until it is repaired. Howevever, this will not cause DEBUG to be delayed. After waiting for the required period it will detect that no reply has been put in NEIGHBOUR and, assuming that a fault is present, it will act accordingly.

A more detailed documentation of FTOS can be found in Appendix 11.

Chapter 5

Fault-Tolerance Implementation

5.1 Assumptions Made

In the previous chapter a set of firmware tests were described. These routines, when performed on the Faulty Processor and interpreted by the Operational Processor, would diagnose certain faults. Throughout the description of this process it was assumed that the CCU was operational. However, by the very nature of the problem no other part of the processor can be assumed to be functioning correctly. In fact the opposite is true, all parts are suspected as being faulty. This poses a problem. Although the CCU is operational it depends upon results of arithmetic operations carried out in other sub-systems. These results are used for directing conditional branch responses in the microcode sequencer (Fig. 5.1). Therefore, there is a significant problem since the CCU cannot rely on the results that it receives. Consequently, it could make erroneous conditional jumps and execute the wrong microcode. The whole processor would effectively be useless and the diagnosis process would be defeated.

In order to counter this situation certain assumptions about the faulty units must be made. These assumptions apply to test results which normally consist of one bit only. If this result is unreliable then there are three possible causes :-

1) It is permanently at the value '1'.

2) It is permanently at the value '0'.

3) It is completely spurious and is intermittently '0' or '1'.

The third possibility is extremely unlikely since most component failures usually cause the output to remain at a certain value. Also, an intermittent or spurious fault is far harder to detect. It was therefore decided to assume one of the first two cases. Again, the choice between these two was based on probability. More likely than not a device will fail completely. In other words, it will be as if its outputs were in the high impedance state. This would normally be interpreted as a '1' by any inputs of another operational device [5.1]. Furthermore, on a practical level, it would be easier to test fault-tolerance firmware if the "permanently at 1" assumption were made. This is because a faulty package could be simulated merely by removing it from its socket.

It was therefore decided that a faulty test result would be assumed to be permanently at 1. Clearly this assumption limits the scope of the diagnosis possible. The author is aware of this fact and suggestions for further research on the problem are included in the final chapter.

5.2 Trapping of Errors

5.2.1 Normal Error Entry

Although a set of tests have been defined to diagnose a faulty processor they will not be of use unless they are called. Therefore, it is required that the CCU should perform regular fault detection.

To a certain extent the structure of P-code is quite advantageous since it requires a good deal of exception checking to be performed at microcode level. Several P-code instructions require checks to be made on the data upon which they are operating. If the data is invalid then an exception will occur. As an example, take the "Test In Set" instruction. This corresponds to the high level "IN" operator in Pascal (as applied to sets). Its function is to take a value off the stack, call this value i. It then checks to see whether the i'th bit of a set (which is also on the stack) is a '1'. Since a set is represented by eight words (128 bits) the value must be in the range 0 <= i <= 127. The "Test In Set" instruction checks this and if the value is out of range an exception is generated. P-code makes many such checks, division by zero and array bounds are further examples.

Therefore, if a processor is faulty it is highly likely that a P-code instruction will either detect an illegal value or incorrectly mark valid data as being in error. This will generate an exception which is merely a jump to the fault diagnosis microcode.

Another way of trapping errors is if an illegal op-code is read from memory. This is also quite likely if the processor incorrectly executes machine code. Illegal op-codes are easily trapped by making all unused values in the Decoder point to the fault diagnosis microcode start address.

Therefore, most failures can be detected either by the presence of invalid data generating an exception or an illegal op-code.

5.2.2 Timer Interrupt Test Routines

Despite the precautions mentioned in the previous sections it is possible for some faults in the processor to go undetected. As an example take the "NOT" P-code instruction. Its function is to invert the word on the top of the stack. No checking of data is required. No arithmetic overflow can occur and the stack pointer is not altered, so stack overflow cannot take place. If the PCU was faulty it might generate the same address at each machine cycle. This means that the processor could become trapped in an infinite loop of the type :-

```
Fetch Instruction,
Instruction = NOT, Execute,
Fetch Instruction,
Instruction = NOT, Execute,
Fetch Instruction,
Instruction = NOT, Execute

etc.
```

Therefore, it is necessary to have some means of identifying errors of this type. Concurrent Pascal requires a record of real-time to be maintained. To achieve this the Super Sixteen generates a timer interrupt at fixed intervals. This means that an infinite loop would be interrupted at some point. It is therefore necessary for the interrupt firmware to perform a test to trap this type of error.

The code to perform this test merely consists of passing values from the PCU via the PCUTRAN to the ALU and checking that they arrive without being corrupted. This test is designed mainly to test the PCU using the ALU. However, if the PCU was functioning correctly and the ALU was faulty then this would also be indicated by the test.

It is possible that a faulty memory, especially an EPROM containing P-code, might generate a similar problem. If, for example, the EPROM was faulty then the processor might pick up every word of the program memory to be the "NOT" instruction. So, to the processor the program would appear as :-

Memory Location	Op-code
x	NOT
x+2	NOT
x+2	NOT
x+6	NOT
.	
.	
etc.	

Which would generate an infinite loop similar to the one previously described.

Therefore, it is necessary for the timer interrupt firmware to check for this condition. The test to detect this is based on a knowledge of the stucture of a compiled Concurrent Pascal program. Referring to Figures 5.2 and 5.3, the compiler always inserts at the start of the program four words of control information followed by a jump past any procedures. Even if there are no procedures this transfer of control is still present. So, assuming that a valid program is stored in EPROM, there will always be a branch instruction in the fifth memory word. It is a fairly simple matter for the interrupt test firmware to read the fifth location and check that it is the op-code for "Jump". If this is not found then the hardware is assumed to be faulty.

To summarise then, most faults are either trapped as illegal op-codes or invalid data in the program by the P-code firmware. Some errors will escape these tests. Hence, it is necessary for a certain amount of testing to be performed by the Timer Interrupt firmware (Fig 5.4).

5.2.3 Faulty Timer Interrupt Hardware

One fault which cannot be detected with the methods described so far is one occuring in in the timer interrupt hardware. If no timer interrupts were being generated it would not affect the P-code in any way, hence no exception would be caused. The only feature of the system that would probably be affected would be the maintenance of real time. It is unlikely that process scheduling would be affected since a Kernel call is made whenever an exit from a Monitor occurs. The solution to the problem lies at the Concurrent Pascal level. Two extra processes must be added to any program. The first process performs a Wait (for one second) operation and then sends a message (via a Monitor) to a second process. The second process maintains a count which is sufficiently large to last for at least one second. If, when the count terminates, no message has been received from the first process then the second one must output a message to indicate that the Timer Interrupt Unit is faulty. This feature of fault detection has not yet been implemented in FTOS.

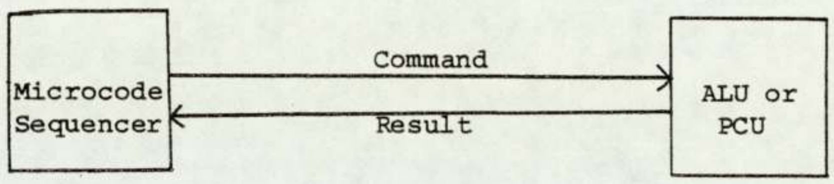


Fig. 5.1. CCU/ALU/PCU Interaction.

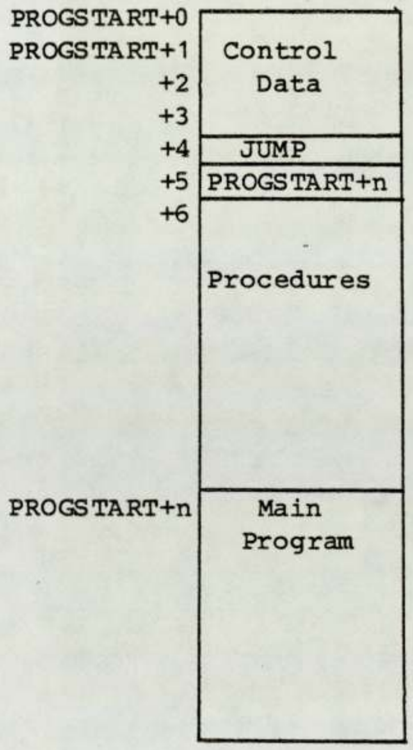


Fig. 5.2. A Typical Concurrent P-code program stored in EPROM.

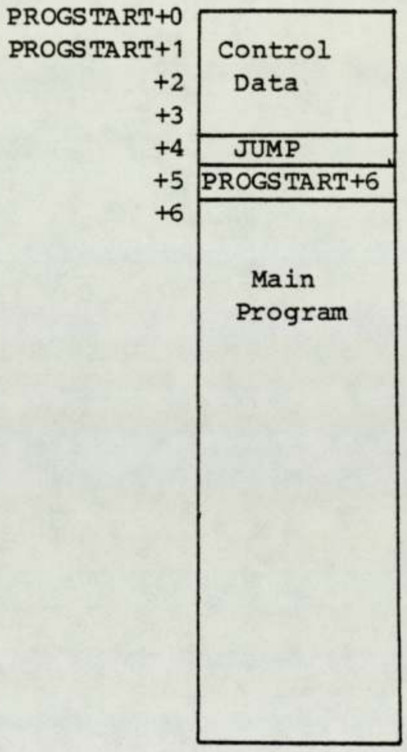


Fig. 5.3. A Concurrent P-code program with no procedures.

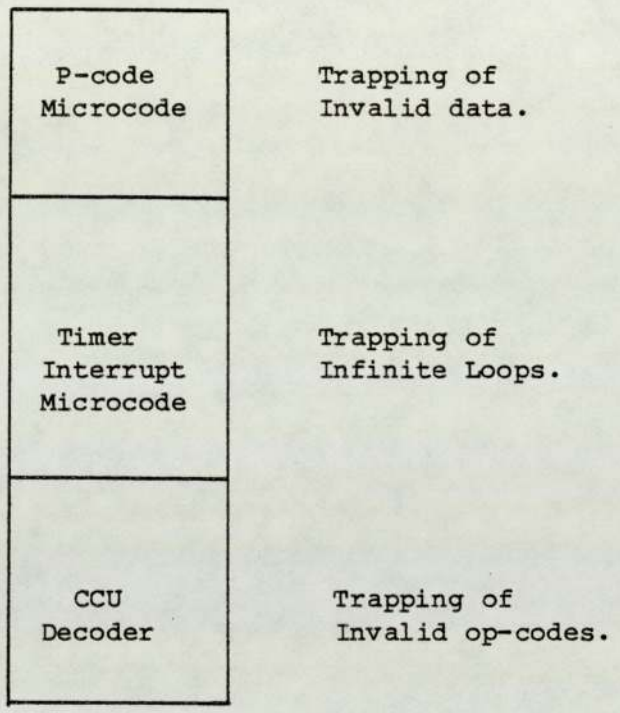

P-code Microcode
Trapping of Invalid data.
Timer Interrupt Microcode
Trapping of Infinite Loops.
CCU Decoder
Trapping of Invalid op-codes.

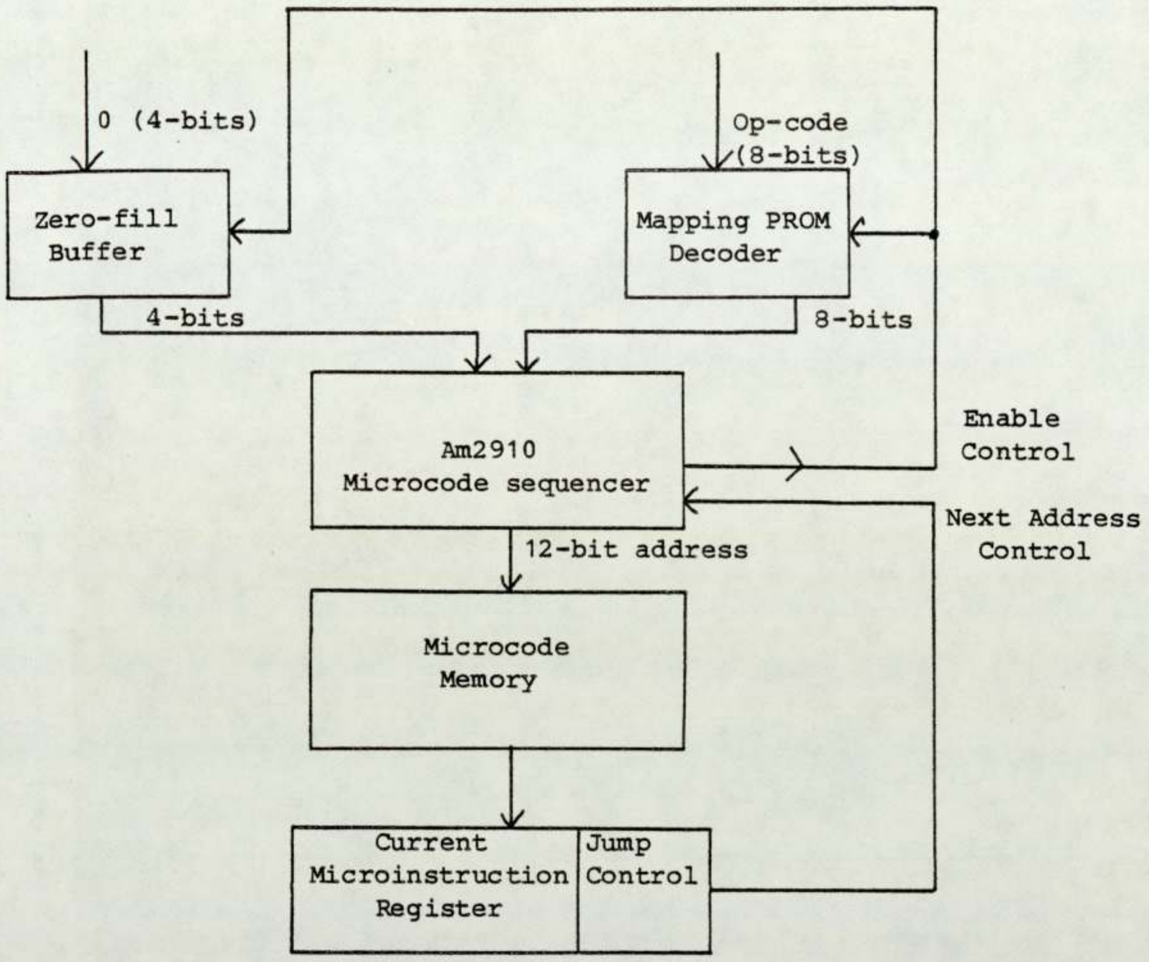
Fig. 5.4. Fault detection in the Super Sixteen

5.2.4 Faulty Mapping PROM Decoder

Another type of fault which would probably always be trapped by one of the methods mentioned in 2.1 or 2.2 occurs when the Mapping PROM Decoder is faulty. The Decoder converts machine instruction op-codes into microcode addresses (Fig. 5.5). It is assumed that the outputs of this 8-bit EPROM would all be '1' if it failed. This means that whenever the CCU executed a "Jump to Map" instruction a branch to address Hex FF would be made. What would happen after this would depend on the microcode stored there. The result is clearly unpredictable, especially when one takes into account that future addition or deletion of microcode to the addresses below Hex FF would alter the code stored in Hex FF itself.

For this reason it was necessary to modify the assembler so that it placed an instruction at Hex FF to enter the fault diagnosis microcode. The assembler always rearranges the instructions in memory and modifies the sequencing fields so that they branch round location Hex FF. This means that if the microinstruction at Hex FF is executed a fault can be assumed. This ensures that a faulty "permanently at 1" Mapping PROM Decoder will be detected.

Before calling the fault diagnosis microcode, a special message is sent to the Operational Processor. In the case of the test sequence $A.B.C.D.E.F.\overline{G}.$ mentioned in Chapter 4, indicating a faulty ZIREG or Decoder, it can resolve which of the two units is faulty. If the Operational Processor receives the message then the Decoder must be functioning correctly and the ZIREG is faulty. If the message is not received then the Decoder is assumed to have failed.



Note

1) A faulty PROM generates 0000 1111 1111 = Hex FF when a "jump to map" instruction is executed.

2) A faulty zero-fill buffer generates 1111 XXXX XXXX = Hex FXX when a "jump to map" instruction is executed (X = DON'T CARE).

Fig. 5.5. Function of the Mapping PROM Decoder.

A faulty zero-fill buffer could be detected by applying the same technique to microcode memory locations \$F00 to \$FFF. However, this would use 256 locations of microcode memory and has not been implemented on the Super Sixteen.

5.3 Microcoding Techniques

5.3.1 Faulty Test Results

The CCU has to contend with the fact that a test result which should be a '0' might be a '1'. If it is executing a loop and is waiting for a test result to terminate it, then that result may never arrive. The CCU would be within an infinite loop. At the heart of the CCU is the Am2910 microprogram sequencer. This device has a one bit input (CC) which determines whether a conditional jump is made. If this bit is zero then the CCU takes the current test (an arithmetic operation in the ALU or PCU) as having passed and a branch is taken. If the bit is a one then the current test has failed and no jump is made.

A faulty unit is assumed to make this test "permanently at 1", hence no conditional jumps will be made. The firmware in the CCU must be structured to take account of this fact. As an example, consider a loop of the following type :-

```
         Load Counter
LOOP:    Decrement Counter, if = 0 jump to NEXT
         microinstruction 1
```

```
        microinstruction 2
          .
          .
        microinstruction n
        Jump back to LOOP
NEXT:   microinstruction n+1
```

A counter is decremented. When it reaches zero an exit is made from the loop using a conditional jump. If the processor is faulty it is possible that this branch is never taken. The CCU would be within an infinite loop. There is obviously no way that a structure of this kind can exist without a conditional transfer of control. To overcome this problem, the loop must be structured so that an exit will be made unless the test succeeds. The solution is to re-write the code as follows :-

```
        Load Counter
LOOP:   Decrement Counter, if NOT = 0 then jump to NEXT1
        Jump to NEXT
NEXT1:  microinstruction 1
        microinstruction 2
          .
          .
        microinstruction n
        Jump to LOOP
NEXT:   microinstruction n+1
```

This stucture will exit the loop if a test fails and remain within it if the test passes. A fault in the ALU test result circuitry would therefore cause an exit to be made.

If the CCU uses test results from the PCU then the situation becomes more complex. Connecting the two sub-systems is a circuit known as the Test Tree (Fig 5.6). It is possible that the Test Tree may be faulty ("permanently at 1") which would cause all CCU tests to fail. Unfortunately, the Test Tree, when operating correctly, inverts all boolean results from the PCU. This means that if the PCU fails ("permanently at 1") the Test Tree will invert the result and make it a zero. All CCU tests would pass.

If the Test Tree is faulty then the previous argument concerning the re-structuring of loops still applies. However, if the PCU itself has failed then this condition must be detected before any potentially infinite loops are entered. This is relatively easy to do. For example, in the PCU memory test described in the previous chapter the first action that is taken is to load one of the PCU registers with the bottom RAM address. This value is always non-zero. It is then possible to structure the code to perform a test using the PCU as follows :-

1) Load PCU register with a non-zero value.

2) Test the PCU register, if it equals zero then the test fails (conditional jump).

3) Perform the Test.

If the PCU itself is faulty then the current test will fail at 2 since all PCU results will cause a conditional branch to be taken. If the Test Tree is faulty then the test will fail at 3) in the manner described for loops previously.

Another problem of this type that occurs is if the most significant device of the 4 X Am2903 bit-sliced ALU is faulty. The four sub-units are cascaded together as shown in Fig. 5.7.

The most significant device is responsible for passing the results of any arithmetic operations to the Am2904 Test Status Multiplexer. This then converts the values into a one bit test result which is conveyed to the CCU. These results consist of 4 bits; the Z flag which is set to 1 if the result of an operation is zero and is zero otherwise, the N flag which is the sign bit, the C flag which is the Carry out and the O flag which indicates overflow. Using these four bits the Test Status Multiplexer can make any boolean comparison (=, NOT =, >, >=, <, <=) for both signed and unsigned values. If the m.s. device is faulty then the N, C and O flags will all be "permanently at 1". There are two facts to be noted about this. First, this does not mean that all tests will automatically pass. For example, the N flag will be taken to be high by the Test Status Multiplexer so that all "less than zero" tests will pass but all "greater than zero" tests will fail. Second, the most significant device is jointly responsible with all the other sub-units for generating the Z bit, since the Z flags from each device are wire-ored together. All of the 4-bit outputs from each package must be '0000' for the Z flag to be high. This means that if the m.s. device has failed the Z bit will be "permanently at 1". However, consider the following case. Suppose a comparison is made. If the two values compared are equal (i.e. the result of a subtraction is zero) then the test being performed passes and fails otherwise. This would be coded as :-

1) Make a comparison, jump to error code if NOT =

2) Test passed, continue

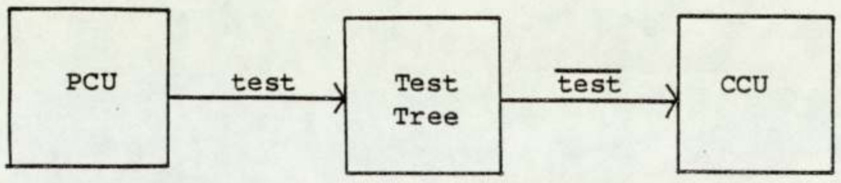


Fig. 5.6. PCU/CCU Interaction

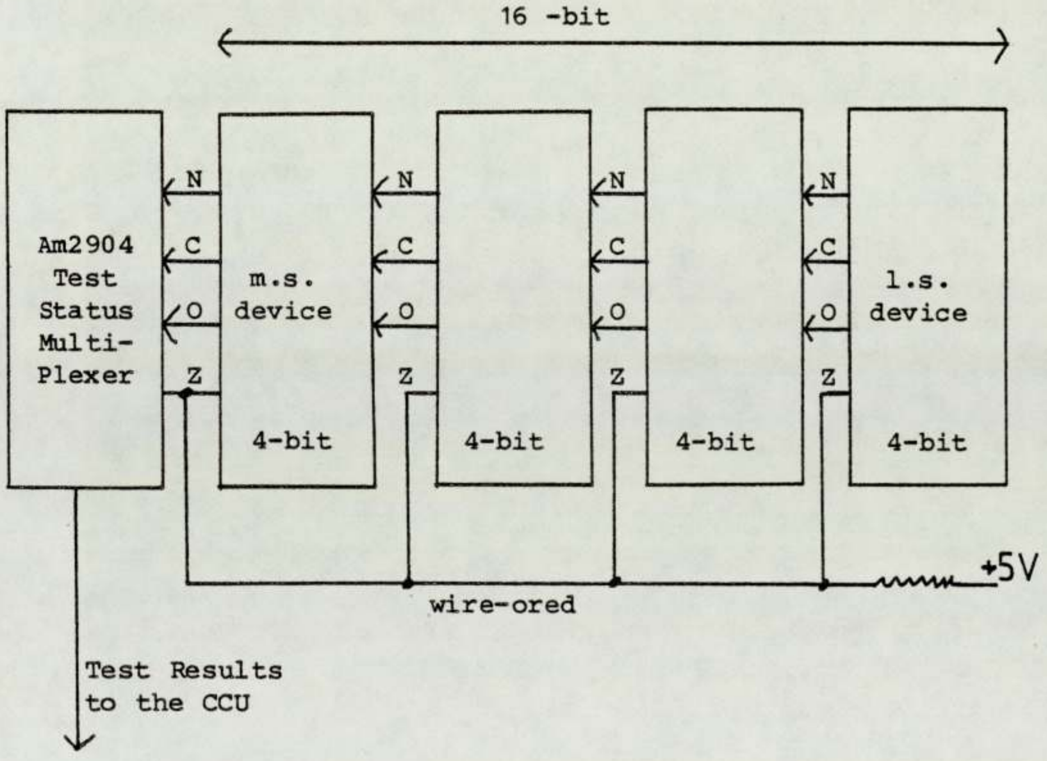


Fig. 5.7. Test evaluation in the bit-sliced ALU.

Suppose the two values being compared are equal, but one of the bit-sliced sub-units other than the m.s. device is faulty. This will mean that the next bit-sliced device in the m.s. direction will receive a '1' as its Carry-in. Hence its four bits will be '0001' rather than '0000', as they should be. The Z flag fed to the Test Status Multiplexer will be low so the test will fail. If the m.s. device fails then its Z flag will be high. This will be wire-ored with the other three Z flags which are also high. The Test Status Multiplexer will assume that the comparison was successful. This would make the above algorithm pass when it should fail. If the methods developed already are used an algorithm would be obtained as follows :-

```
        Make comparison, if = jump to NEXT
        Jump to error
NEXT    Continue with the rest of the test
```

This algorithm would also fail since if the m.s. ALU device were faulty then no error would be detected. So, an algorithm must be developed such that if two equal values were compared and the m.s. ALU sub-unit were faulty, then all test flags (Z,N,C and O) would be "permanently at 1". The solution to this problem is quite simple, a value cannot be zero and negative, hence if N and Z are both high there is an error. The algorithm, therefore, is as follows :-

```
1) Make Comparison, error if NOT =
2) Make Comparison (again) if result is -ve then error
3) Continue with the rest of the test
```

The Am2904 Test Status Multiplexer is also responsible for other functions. One of these is the generation of a Carry-in bit for the ALU (Fig. 5.8). This value can come from several sources. Typically, it would be '0' if a value were being loaded into the ALU or an addition were taking place and a '1' if a subtraction were being performed. However, if the Am2904 were faulty the Carry-in to the ALU would always be high. This means that when loading constant values into the ALU a failure of the Am2904 would cause the message to be corrupted. This is overcome by ensuring that during assembly such constants are decremented by one with the Carry-in value set to a '1'. This means that it does not matter whether the Carry-out is operational or "permanently at 1". Addition and loading of non-constant values (i.e. data read from memory etc.) will still be incorrect. However, this will only cause tests to fail whereas invalid constants would result in incorrect messages being sent to the Operational Processor.

5.3.2 Message Transmission

It is vitally important whenever a test succeeds that a message is sent to indicate this result. Messages are sent via Peripheral Interface Adapters (PIAs). The normal method of accessing a PIA is by putting its address in the MAR and reading or writing a value from or to it via the ALU. If a successful test has been performed by certain sub-systems then those are the only functional units that can be assumed to be operational. This means that the CCU should depend on no other units to send a message to the Operational Processor. It is therefore necessary to have three sets of microcode; one section of code to send a message (and receive a reply) using the ALU only, another using the PCU only and a third employing both the ALU and the PCU but without using the PCUTRAN.

This means that the microcode for sending messages and receiving replies is very tedious and repetitive. It is performing the same task in three different ways. This is a necessary but inelegant requirement. However, this is a form of firmware redundancy. A faulty sub-system only prevents any messages generated by that unit from being sent. The Operational Processor can assume that any messages that are not received are due to the failure of the corresponding test. This duplication of code does ensure that successful test results produced by error-free sub-systems will be sent to the Operational Processor.

Another deviation from the standard practice of sending messages via a PIA is required. Normally a PIA is initialised at the start of a power-up sequence in the processor. Thereafter it can be accessed whenever required. This initialisation sequence consists of writing values to the control and other registers in the PIA. It is possible that a faulty sub-system may try to access these PIAs and corrupt them. This could mean that an error-free unit might subsequently be unable to send an important message. It is therefore necessary for the PIAs to be re-initialised by the sub-system before it sends a message. Again, it is necessary to have three pieces of microcode to initialise the PIAs using different functional units to perform the task.

It is also possible that a value read from the PIA is corrupted en route to the testing unit. If a message is sent and a reply does not arrive, then no further action can be taken other than to provide a backup set of PIAs. This is discussed in Chapter 6. However, it is possible that the fault is in the Datapath and is a read-only fault. This would mean that messages are being sent but replies are not being received. It is important that the Faulty Processor should wait for a reply. If it sends two messages in too short a period then one may be lost due to the receiving processor not having time to process both messages. This means that in addition to having three separate pieces of code for sending and receiving messages, it is necessary for each of them to check the integrity of the data received. These tests are performed by examining the PIA control register (Fig. 5.9) [5.2]. The six least significant bits of this register should remain constant unless written to by the processor. The only bits that can possibly alter are the two m.s. ones which indicate whether replies to messages have been received. It is a simple matter to examine and test the six l.s. bits for comparison. A difficulty occurs when the PCU performs this test, since it has no 'AND' facility to mask out the two m.s. bits. This means that four comparisons have to be made corresponding to each permutation of the two bits.

Once a fault in the PIA read mechanism has been detected a method must be devised to ensure that another message is not sent until the Operational Processor has processed the message. This is done by entering a delay routine. Unfortunately it is not posssible to use the ALU or the PCU to perform a count since they might well be faulty. The Am2910 microcode sequencer posseses an internal counter. This can be used to implement a delay subroutine entirely within the CCU. This counter is only 12 bits wide and, in practice, the delay needs to be greater than 4096 (2 to the power of 12). This means that the microcode must be tediously structured as

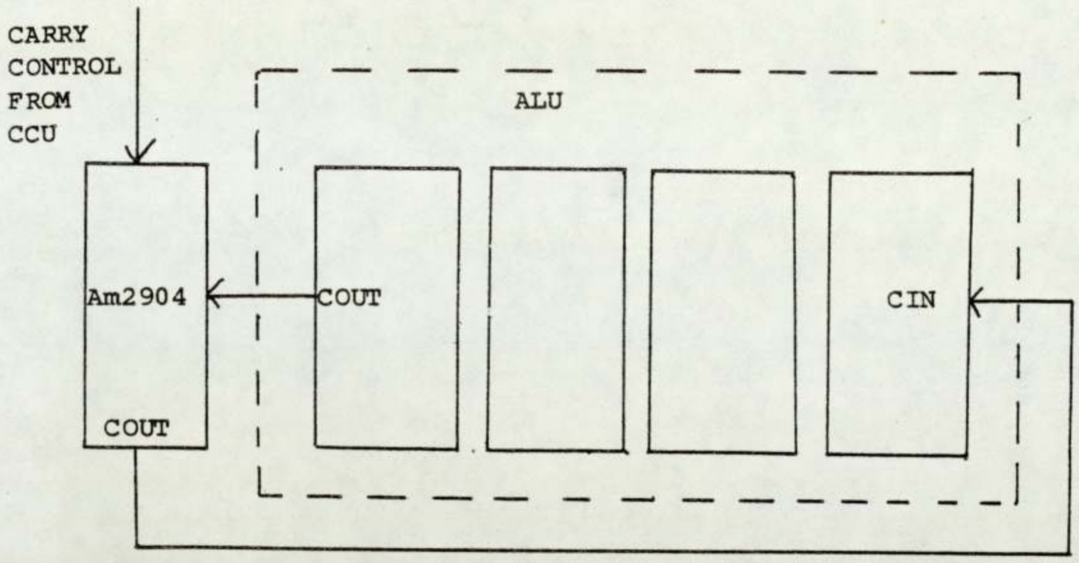


Fig. 5.8. Use of the Am2904 Test Status Multiplexer to generate the ALU Carry-in.

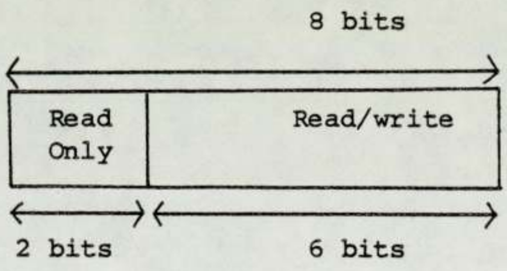


Fig. 5.9. PIA Control Register.

follows :-

```
       Load counter with x
       Decrement counter, repeat if NOT = 0
       Load counter with x
  n    Decrement counter, repeat if NOT = 0
times     .
          .
          .
       Load Counter with x
       Decrement counter, repeat if NOT = 0
```

The above will obtain a delay for n*(x+1) microcycles.

5.4 Summary

Several aspects of writing fault-tolerant microcode have been outlined in this chapter. It is necessary for the CCU to assume that faulty components will remain permanently at a certain binary value. The "permanently at 1" case is the most likely and also the easiest to simulate.

In order to be able to detect all fault conditions it is necessary to perform fault checking during the timer interrupts. However, the structure of P-code is such that many errors will be detected before an interrupt occurs.

The CCU (which is assumed to be operational) depends upon test results which may be faulty. Therefore, it is necessary to structure the microcode in such a fashion that the CCU cannot remain within an infinite loop.

There are also problems associated with sending messages and receiving replies from the neighbouring processor. The solution to this involves firmware redundancy at the cost of inelegance of the microcode. This ensures that successful test messages are sent using only error-free units.

Chapter 6

Conclusions and Proposals for Further Work

6.1 Conclusions

The fault-tolerant, bit-sliced microprocessors described in this thesis have been built and tested. With the exception of certain hardware problems within the processor implementation, the fault diagnosis and recovery procedures function exactly as predicted. At times, the hardware was susceptible to external noise. This can be a problem with a large, wire-wrapped, prototype system.

The system was tested by removing packages from their sockets to simulate a fault. The diagnosis process would start and the Operational Processor would take over all system tasks. The integrity of the database was maintained by periodically updating the backup processor. Hence, a set of principles have been derived and put into practice. These could be adapted to implement any fault-tolerant, real-time system using bit-sliced components. Therefore, these principles are appropriate to all types of computer.

The intentions of this project were to design and verify a fault-tolerance philosophy using bit-slice techniques. To this extent it has been successful. Also, a useful by-product has been that a fast and efficient Concurrent Pascal machine has been produced on which small operating systems can easily be written.

It is interesting to constrast this project with [1.18] as mentioned in Chapter 1. The approaches differ in two respects. First, the Super Sixteen processor contained no special-purpose fault-tolerant hardware. The firmware merely demonstrated the inherent redundancy within the machine. Second, the firmware was capable of diagnosing some faults within sub-systems other than those which used the actual bit-sliced devices. For example, the Datapath and the PCUTRAN.

It is significant to note that a large amount of redundancy exists within the hardware of a fast processor such as the Super Sixteen. Only the firmware can utilise this redundancy. A level of fault-tolerance within a single processor has been achieved which is considerably greater than that which could be achieved by software alone. To achieve a similar level of fault-tolerance using software, it would be necessary to augment the hardware.

Fault-tolerance firmware provides a half-way stage between no fault tolerance and a full TMR system. If there is a need to improve the processor reliability without incurring the cost of a TMR system, firmware techniques may be appropriate. Of course, this implies the use of a microprogrammable machine. It would be possible to implement fault-tolerance firmware on an existing machine by the addition of appropriate microcode.

6.2 Suggestions for Further Research

Whilst engaged on this research project several ideas for further research in this field have occured to the author. These suggestions are divided into two categories. The first section contains improvements to the existing system. These are extensions to the work carried out to date but have not been pursued due to the limited amount of time available.

6.2.1 System Improvements

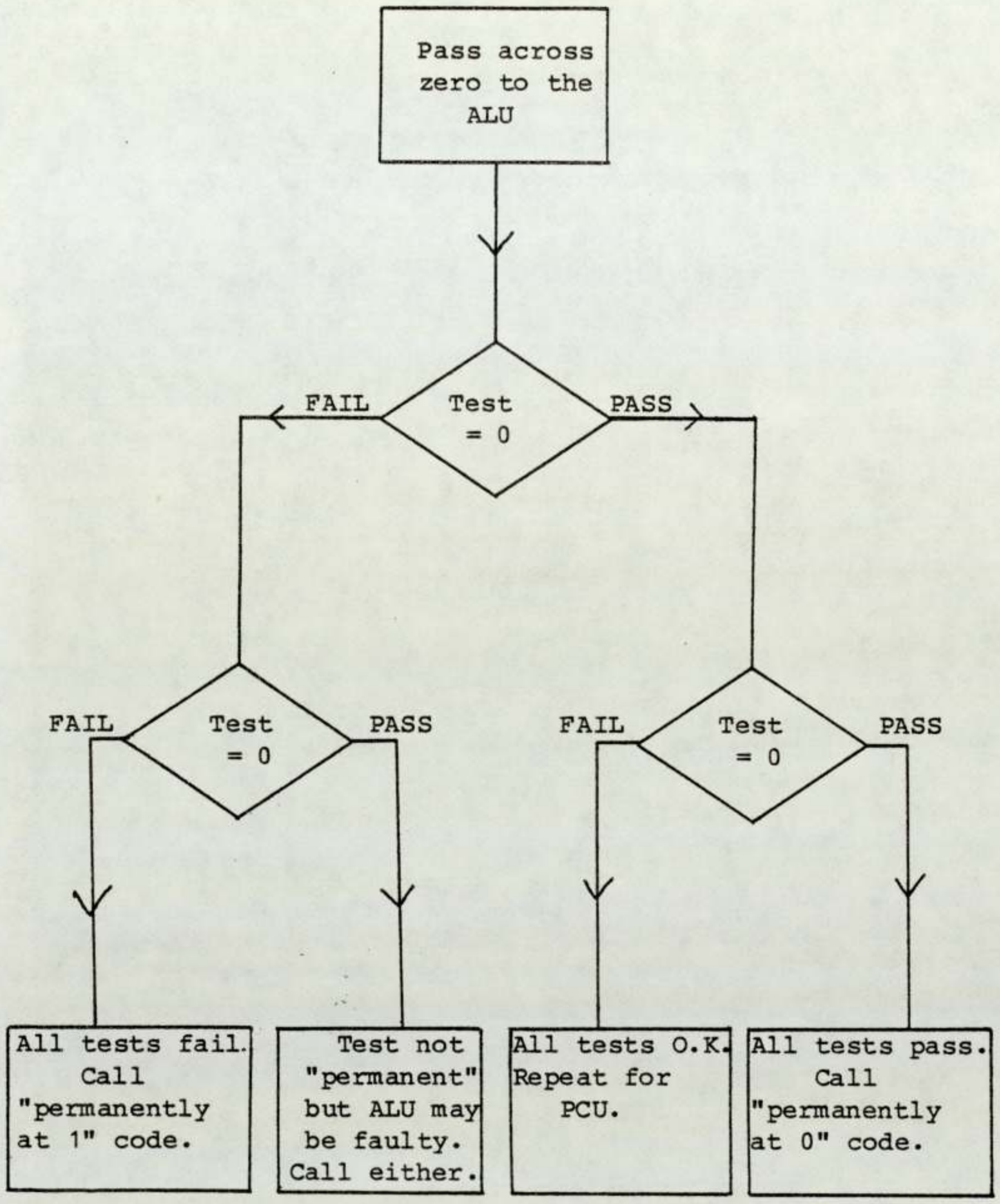
The system that has been built demonstrates the principles of achieving fault-tolerance through the use of bit-slice techniques. However, it could be improved as a working unit.

To enhance the operating speed of the processor and minimise its susceptibility to switching noise it would be necessary to re-design the board layout.

Although the Super Sixteen can diagnose a fault in the PCU, it cannot at present diagnose which of the four bit-sliced devices making up that sub-system has failed. It would be a reasonably simple matter to remedy this situation. The CCU would need to transmit values through the unit and then forward them to the Operational Processor as a message. The Operational Processor would determine which one of the four 4-bit fields was corrupted.

It was assumed throughout Chapter 5 that a faulty device would produce binary outputs of '1'. The reasons for this assumption were given at the time, but it was stated that they were based on probability. It would be highly desirable for the system to be able to detect "permanently at 0" faults. This could be achieved by having two sets of microcode. To supplement the present code there could be a similar set of tests but they would assume the "permanently at 0" case. At the start of the diagnosis sequence the CCU would have to determine which set of tests to choose (Fig. 6.1). This could be achieved in a similar manner to that used to solve the problem with the PCU and the Test Tree described in Section 5.3.1. In other words the CCU would pass the value '0' through the ALU and then test to see if it had been altered. It would then re-test to see if it were non-zero. If both tests failed the "permanently at 1" microcode would be called. If they both passed then the "permanently at 0" code would be executed. If the results were both correct then the process would be repeated on the PCU to determine whether it was permanently at a certain value. As in previous examples, this is a form of microcode redundancy and is gained at the cost of elegance and compactness in the firmware.

Because the Super Sixteen uses memory-mapped I/O a problem might arise. This would be if the memory board either failed completely or in such a way as to make the PIAs inaccessible. The Faulty Processor would be functioning correctly but would be unable to send any messages to its operational neighbour. The memory-mapped I/O facility is required in order to make test G in Chapter 4 possible. However, a set of backup PIAs, which were not memory mapped, could be provided. In other words, the processor would have direct access to these PIAs. They could be connected to the YBUS of the Super Sixteen. This would give both the ALU and the PCU access to them.



Pass across zero to the ALU
FAIL
Test = 0
PASS
FAIL
Test = 0
PASS
FAIL
Test = 0
PASS
All tests fail. Call "permanently at 1" code.
Test not "permanent" but ALU may be faulty. Call either.
All tests O.K. Repeat for PCU.
All tests pass. Call "permanently at 0" code.

Fig. 6.1. An algorithm for deciding whether to call "permanently at 0" microcode or "permanently at 1" microcode.

The practical work undertaken on this project has concentrated on faults within the processor. If the main memory circuitry fails then the Operational Processor is only aware of this basic fact. It has no information concerning which part of the memory has failed. The memory tests performed could be enhanced so that they determine the faulty memory addresses. A test could also be performed to try and detect whether the fault was in the most significant 8-bit memory board or the least significant one.

6.2.2 Further Research

At present the CCU is a functional sub-system whose failure would prove catastrophic to the fault diagnosis of the system. The first two suggestions in this section are concerned with remedying this weakness.

Microcode memory consists of various fields of control data. In a normal microcoded machine the logical format of these fields would take no consideration of the physical boundaries of the ROMs on which they were stored. A set of fields controlling a sub-system might be scattered across several physical devices. A failure of any one of these, or the latches constituting the pipeline register, would cause all the units they were part-controlling to become inoperable. There would be no overhead or disadvantage in formatting these fields so that they occupied a single, or at most two, ROMs. No single physical device would hold code controlling more than one sub-system. This would mean that a fault in a single memory device within the CCU would only cause one sub-system to fail (Fig. 6.2). This fault would be diagnosed in the normal manner.

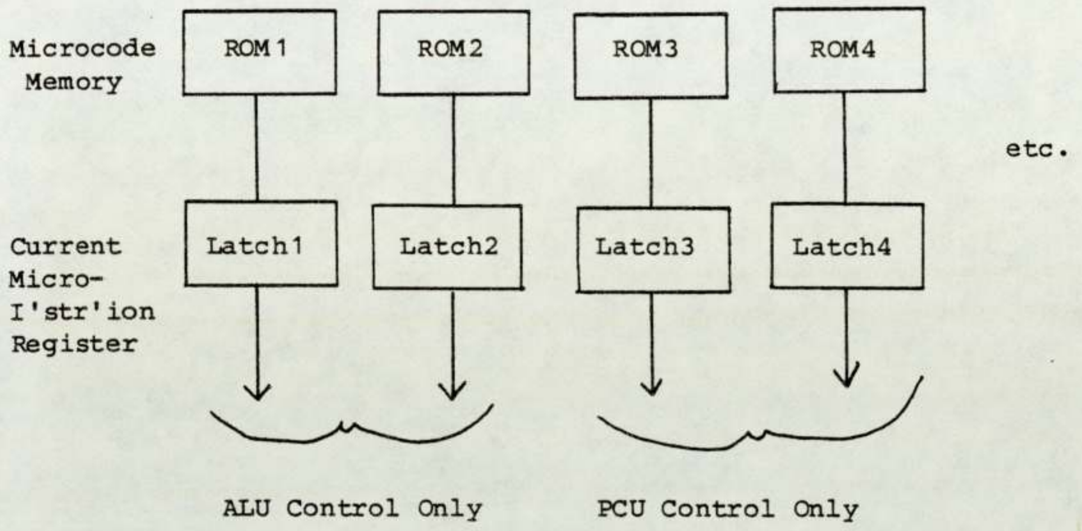


Fig. 6.2. Partitioning of microcode in memory to achieve greater reliability of the CCU.

When repairing the sub-system the engineer would check that the ROM and latch controlling that sub-system were not faulty.

The second suggestion for improving the CCU reliability is to use self-checking hardware. The microprogram sequencer could be modified by duplicating the sequencer and introducing a self-checking checker circuit to compare the outputs. This approach has already been taken in [1.18] and was mentioned in Chapter 1. The microcode memory and pipeline register could either be checked using the method of partitioning the microprogram fields, as mentioned above, or by using parity which has been employed in [1.18]. A fault-tolerant system clock could also be introduced. This could be achieved by using similar redundancy of hardware.

If the CCU was totally self-checking and the Datapath devices were duplicated then the entire processor could be completely self-diagnosable for any single unit failure. The cost of the extra hardware compared to the total processor cost would be minimal, particularly when compared to some of the systems described in Chapter 1. If the CCU and the clock control circuitry were to use some form of Triple Modular Redundancy then the processor should be able to function to some extent, although at a slower rate, when any single fault is present.

Several papers have been published on the pure mathematical theory of fault diagnosis [6.3 - 6.12]. These are all based on an original paper by Preperata et al. [6.2]. The theory investigates how many units can fail and still be diagnosed correctly by other operational ones within a system. The ALU/PCU analysis within the Super Sixteen can be used to draw an analogy with this theory.

There are three units; the ALU, the PCU and the PCUTRAN. If any one of them fails then the other two can effectively diagnose the faulty unit. It would be interesting to investigate how this theory could be applied in terms of bit-slice machines. It would be useful to determine whether the addition of further arithmetic units within a single processor would improve its reliability. It might be appropriate to use simulation in such an investigation. It would also be a logical development to determine whether multiple sub-system failures could be diagnosed.

Another idea which merits further thought is the reconfiguration of the processor architecture at microcode level. If it were still possible for the processor to operate at a satisfactory speed then a reconfiguration could take place under certain circumstances. Normally, the microcode will make full use of all the facilities within the architecture available to it. However, under degraded conditions it could continue to function without using certain components. There is no reason why the CCU could not have several Decoders pointing to different blocks of microcode. Only one of these would be selected at any particular time. One Decoder would address the standard microcode. Another one would address a block of microcode which executed machine instructions without using the ALU, this would be selected in the event of its failure. There would be another block of code which did not use the PCU and also a section which used byte memory addressing only.

Such a reconfigurable capability would require a significant duplication of microcode and would use large amounts of ROM. However, it would be possible provided that the speed of processing was still satisfactory. This would allow the processor to continue functioning under degraded conditions. This might be useful in a medium to large multiprocessor system. The number of tasks performed by a faulty processor could be reduced but not entirely eliminated. It would still contribute to the running of the system, its individual performance would merely be degraded.

References

[1.1] IEEE Computer Society
International Symposium on Fault-Tolerant Computing, 1971.

[1.2] IEEE Computer Society
International Symposium on Fault-Tolerant Computing, 1972.

[1.3] Martin, J.
Design of Real-Time Computer Systems, Prentice Hall Series in Automatic Computation, pp. 56-61.

[1.4] Nissen, J.C.D. and Geiger, G.V.
A Fault-Tolerant Multimicroprocessor for Telecommunication and General Applications, G.E.C. Telecommunications Ltd., P.O. Box 53, Coventry.

[1.5] Rennels, D.A.
Reconfigurable Modular Computer Networks for Spacecraft On-Board Processing, Computer, July 1978, pp. 49-59.

[1.6] Enslow, P.H.
Multiprocessor Organisation - A Survey, Computing Surveys, Vol. 9, No. 1, March 1977, pp.103-128.

[1.7] Hamer-Hodges, K.J.
Fault Resistance and Recovery within System 250, Proc. ICCC Conf. on Telephone Systems, 1972, pp. 290-295.

[1.8] Cosserat, D.C.
A Capability Oriented Multi-Processor System for Real-Time Applications, Proc. ICCC Conf. on Telephone Systems, 1972, pp. 282-289.

[1.9] Repton, C.S.
Reliability Assurance for System 250, A Reliable, Real-Time Control System, Proc. ICCC Conf. on Telephone Systems, 1972, pp. 297-305.

[1.10] Avizienis, A., Gilley, G.C., Mathur, F.P., Rennels, D.A., Rohr, J.A. and Rubin, D.K.
The STAR (Self-Testing and Repairing) Computer: An Investigation into the Theory and Practice of Fault-Tolerant Computer Design, IEEE Transactions on Computers, vol. C-20, No. 11, Nov. 1971, pp. 1312-1321.

[1.11] Hopkins, A.L.
A Fault-Tolerant Information Processing Concept for Space Vehicles, IEEE Transactions on Computers, Nov. 1971, pp. 1394-1403.

[1.12] Wakerly, J.F.
Microcomputer Reliability Improvement Using Triple-Modular Redundancy, Proceedings of the IEEE, Vol. 64, No. 6, June 1976, pp. 5-1 to 5-7.

[1.13] Carter, W.C. and Schneider, P.R.
Design of Dynamically Checked Computers,IFIP Congr. 1968, Vol. 2, 1968,pp. 878-883.

[1.14] Anderson, D.A. and Metze, G.
Design of Totally Self Checking Check circuits for m-out of n Codes, IEEE Transactions on Computers, Vol. C-22, March 1973, pp. 263-269.

[1.15] Abd-alla, A.M. and Meltzer, A.C.
Principles of Digital Computer Design, Vol. 1, Prentice Hall, 1976, pp. 83-89.

[1.16] Reddy, S.M.
A Note on Self-Checking Checkers, IEEE Transactions on Computers, Vol. C-23, Oct. 1974, pp. 1100-1102.

[1.17] Diaz, M.
Conception de Systemes Totalement Auto-Testables et a Pannes Non-Dangereuses, Ph.D. Thesis (Universite Paul Sabatier de Toulouse) no. 618, 1974.

[1.18] Ciompi, P. and Simoncini, L.
Design of Self-Diagnosable Minicomputers Using Bit-Sliced Microprocessors, Journal of Design Automation and Fault-Tolerant Computing, Vol. 1, No. 4, Oct. 1977, pp. 363-375.

[1.19] Katzan, H.
Microprogramming Primer, Mcgraw Hill, 1977.

[1.20] PR1ME Computer Co. Ltd.
PR1ME Microcoders Handbook, 145 Pennsylvania Ave., Framingham, Mass. 01701, USA, pp. C-11 to C-14.

[2.1] Woodward, P.M., Wetherall, P.R. and Gorman, B.
Official Definition of Coral 66, HMSO, London, 1973.

[2.2] Barnes, J.G.
RTL/2 Design and Philosophy, Heyden and Son Ltd., 1976.

[2.3] Wirth, N.
Modula: A Language for Modular Multiprogramming, Software Practice and Experience, Vol. 7, No. 1, 1977, pp. 3-35.

[2.4] Wirth, N.
The Use of Modula, Software Practice and Experience, Vol. 7, No. 1, 1977, pp. 37-65.

[2.5] Wirth, N.
Design and Implementation of Modula, Software Practice and Experience, Vol. 7, No. 1, 1977, pp. 67-84.

[2.6] Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Rouline, O. and Wichmann, B.A.
Ada Preliminary Reference Manual and Design Rationale, SIGPLAN, Special Edition, June 1979.

[2.7] Brinch-Hansen, P.
The Programming Language Concurrent Pascal, IEEE Transactions on Software Engineering, June 1975, pp. 199-207.

[2.8] IEEE Computer Society.
Stack machines, special edition, Computer, May 1977.

[2.9] Brinch-Hansen, P.
The Architecture of Concurrent Programs, Prentice Hall, 1977.

[2.10] Graef, N., Kretschmar, H. and Loehr, K. and Morawetz, B.
How to Design and Implement Small Time-Sharing Systems using Concurrent Pascal, Software Practice and Experience, Vol. 9, 1979, pp. 17-24.

[2.11] DAHL, O.J., DIJKSTRA, E.W. and Hoare, C.A.R.
Structured Programming, Academic Press, New York, 1972.

[2.12] Matison, S.E.
Implementation of Concurrent Pascal on LSI-11, Software Practice and Experience, Vol. 10, 1980, pp. 205-217.

[2.13] Kerridge, J.M.
A Fortran Implementation of Concurrent Pascal, Software Practice and Experience, Vol. 12, 1982, pp. 45-55.

[2.14] Hall, J.A.
A Microprogrammed Interpreter for the Data General Eclipse S/130 Minicomputer, Software Practice and Experience, Vol. 12, No. 8, pp. 755-765.

[2.15] Neal, D. and Wallertine ,V.
Experiences with the Portability of Concurrent Pascal, Software Practice and Experience, Vol. 8, 1978, pp. 341-353.

[2.16] Chattergy, G.
Microprogrammed Implementation of a Scheduler, Sigmicro Newsletter, Vol. 7, Sept. 1976, pp. 15-19.

[3.1] Advanced Micro Devices Ltd.
The Am2900 Family Data Book, 901 Thompson Place, Sunnyvale, California 94086, USA, 1979.

[3.2] Mick, J. and Brick, J.
Bit-Slice Microprocessor Design, McGraw Hill, 1980.

[3.3] Stuart James Systems Ltd.
6809 Microcomputer Documentation, Stuart James Systems Ltd., 16 Watling street, Wall, Lichfield, Staffs.

[5.1] Lenk, J.D.
How to Troubleshoot and Repair Microcomputers, Repton Publishing, pp. 213-217.

[5.2] Motorola Semiconductors Products Inc.
M6800 Microprocessor Applications Manual, pp. 3-8 to 3-20, Motorola Semiconductor Products Inc., York House, Empire Way, Middlesex.

[6.1] Wilkes, M.V. and Stringer, J.B.
Microprogramming and the Design of Control Circuits in an Electronic Digital Computer, Proceedings of the Cambridge Philosophical Society, Part 2, Vol. 49, April 1953, pp. 230-248. Reprinted in Computer Structures: Readings and Examples, C.G. Bell and A. Newell, McGraw Hill, New York, 1971.

[6.2] Preperata, F.P., Metze, G. and Chien, R.T.
On the Connection Assignment of Diagnosable Systems, IEEE Transactions on Electronic Computers, Vol. EC-16, Dec. 1967, pp. 848-854.

[6.3] Preperata, F.P.
Some Results on Sequentially Diagnosable Systems, Proceedings of the Hawaii International Conference on System Sciences, University of Hawaii, Jan. 1968, pp. 623-626.

[6.4] Hakimi, S.L. and Amin, A.T.
Charicterisation of the Connection Assignment of Diagnosable Systems, IEEE Transactions on Computing, Vol. C-23, Jan. 1974, pp. 86-88.

[6.5] Ciompi, P. and Simoncini, L.
The Boundary Graphs: An Approach to the Diagnosability With Repair of Digital Systems, Proceedings of the third Texas Conference on Computer Systems, Austin, pp. 9.3.1-9.3.9, Nov. 1974.

[6.6] Ciompi, P. and Simoncini, L.
On the Diagnosability With Repair of Digital Systems, Technical Report no. 75001/E, Pisa, Convenzione Selenia - CNR, Jan. 1975.

[6.7] Friedman,A.D.
A New Measure of Digital System Diagnosis, International Symosium of Fault-Tolerant Computing, FTC-5, Paris, Jan. 1975, pp. 167-171.

[6.8] Barsi, F., Grandoni, F.and Maestrini, P.
Diagnosability of a System Partitioned into Complex Units, International Symposium on Fault-Tolerant Computing, FTC-5, Paris, pp. 171-176, June 1975.

[6.9] Barsi, F., Grandoni, F, and Maestrini, P.
A Study on Self-Diagnosis of Digital Systems, Technical Report, no. 75003/E, Pisa, Convenzione Selenia - CNR, Feb. 1975.

[6.10] Russel, J.D. and Kime, C.R.
System Fault Diagnosis: Closure and Diagnosability with Repair, IEEE Transactions on Computers, Vol. C-24, Dec. 1975, pp. 1155-1161.

[6.11] Russel, J.D. and Kime, C.R.
System Fault Diagnosis: Exposure and Diagnosability without Repair, IEEE Transactions on Computers, Vol. C-25, Jan. 1976, pp. 228-236.

[6.12] Maheshwori, S.N. and Hakimi, S.L.
On Models For Diagnosable Systems and Probabilistic Fault Diagnosis, IEEE Transactions on Computers, Vol. C-25, Jan. 1976, pp. 228-236.