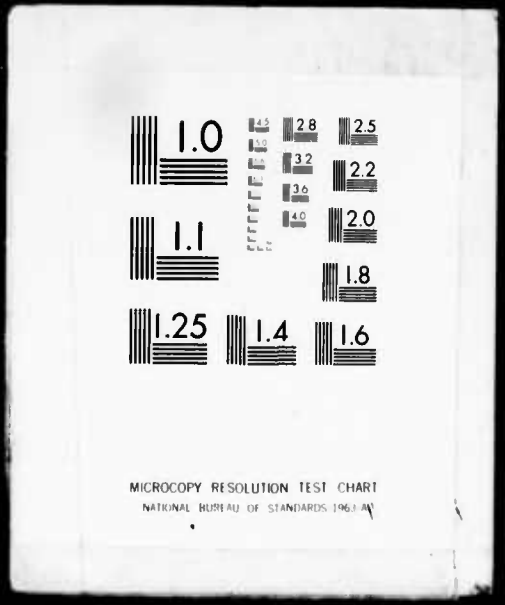
AD-A103 438 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
A PRELIMINARY TESTABILITY ANALYSIS OF THE MIL-STD-1862 ARCHITEC--ETC(U)
AUG 81 F M SMITH, J A BANNISTER DAAK80-79-C-0780
UNCLASSIFIED RTI/1822/04-02F CECOM-80-0780-F NL

END
DATE FILMED
10-81
DTIC



LEVEL II

12

AD A103438

RESEARCH AND DEVELOPMENT TECHNICAL REPORT

CECOM - 80-0780-F

A PRELIMINARY TESTABILITY ANALYSIS OF THE MIL-STD-1862 ARCHITECTURE

F. M. Smith
J. A. Bannister
Research Triangle Institute
P.O. Box 12194
Research Triangle Park, NC 27709

August 1981

Final Report for period 22 May 1979 to 31 December 1980

DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

Prepared for:
CENTER FOR TACTICAL COMPUTER SYSTEMS

CECOM
U S ARMY COMMUNICATIONS-ELECTRONICS COMMAND
FORT MONMOUTH, NEW JERSEY 07703

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CECOM-80-0780-F	2. GOVT ACCESSION NO.: AD-A103 438
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): A Preliminary Testability Analysis of the Mil-STD-1862 Architecture.	5. TYPE OF REPORT & PERIOD COVERED: Final Report, 22 May 79 - 31 Dec 80
	6. PERFORMING ORG. REPORT NUMBER: RTI/1822/04-02F
7. AUTHOR(s): F. M. Smith, J. A. Bannister	8. CONTRACT OR GRANT NUMBER(s): DAAK80-79-C-0780
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Research Triangle Institute, Systems and Measurements Division, P.O. Box 12194, Research Triangle Park, NC 27709	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Task 4.0
11. CONTROLLING OFFICE NAME AND ADDRESS: Test, Measurement & Diagnostic Systems Division (CENTACS), U.S. Army Communications-Electronics Comm., Fort Monmouth, NJ 07703, DRSEL-TCS-MS	12. REPORT DATE: August 1981 13. NUMBER OF PAGES: 81
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Test, Measurement & Diagnostic Systems Division, U.S. Army Communications-Electronic Comm., Ft. Monmouth, NJ 07703	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release
Distribution Unlimited

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Built-in-Test, Exceptions, Interrupts, Rollback-an-Recovery, Retry, ADA

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This study has addressed the ramifications of built-in-test (BIT) as an integral part of the military computer family (MCF) architecture. This was done by looking at concurrent and nonconcurrent BIT and how it would fit into the current MCF architecture specifications.

The current reporting mechanisms in the MCF architecture were evaluated to see which would best serve as a reporting mechanism for concurrent BIT signals.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

A 411039

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

As an upshot of error detection, an error recovery strategy is proposed. As a consequence of this comprehensive recovery strategy, a set of instructions are proposed that would aid in error recovery.

Test and recovery in software is the main thrust of the nonconcurrent BIT section of this study. An error "data base" is proposed. This data base could be accessed as a history by maintenance personnel to provide information to an intelligent error handler and to provide information for reconfiguration control. Several instructions are proposed for doing fault diagnosis and isolation. Rollback and recovery is discussed along with the concept of a recovery cache.

PREFACE

This report was prepared for the Test Measurement and Diagnostic Systems (TMDS) Division of the U.S. Army Communications Research and Development Command (CORADCOM). The Project technical monitor was George Burbank of TMDS.

This report was prepared by F. M. Smith and J. A. Bannister of the Digital Systems Research Section, Systems Engineering Department of the Research Triangle Institute.

The authors wish to thank William Dietz, Leland Szeweranko and Mario Barbacci, of Carnegie-Mellon University, for many interesting and informative discussions about the MIL-STD-1862 architecture.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The Military Computer Family (MCF) concept calls for the government to relinquish specific implementation control and specify only form, fit, and function (F^3) requirements [7]. As a consequence, an area of concern is that of built-in-test (BIT), a critical component of the MCF maintenance concept [1,2]. Because of the F^3 procurement approach, BIT is specified by stating requirements in terms of "percentage of failures detected" rather than in terms of specific BIT techniques. For example, the AN/UYK-41 member of the Military Computer Family, has a fault detection objective of 98 percent with less than a 1 percent false alarm rate.

In previous studies of self-test approaches for MCF, RTI identified BIT mechanisms and their corresponding fault manifestations [3,4]. In this earlier work, RTI studied the effects of faults on software program behavior [4]. The approach taken was to describe an implementation of the existing MCF architecture, PDP-11/70, using the Instruction Set Processor language (ISP). The ISP description was simulated and faults injected [5]. Selected test programs were run using simulation and the impact of these faults on the software observed. The resulting fault manifestations were characterized and their cause and effect relationships analyzed.

The initial PDP-11/70 architecture, has been superseded by a new 32-bit architecture defined by MIL-STD-1862 [6]. The new architecture has not yet been implemented, so now is an appropriate juncture for a critical analysis of its predicted testability characteristics. Modifications and additions to existing MIL-STD-1862 features should be incorporated as early as possible in the development process in order to ensure the testability of future implementations.

This report discusses MIL-STD-1862 built-in-test and the implications of BIT for the software. This work is a logical follow-on to RTI's previous work on BIT approaches for detecting errors and handling these errors in MCF machines.

Built-in-Test approaches discussed in this report fall into two categories; concurrent and nonconcurrent BIT (as shown in Figure 1 and discussed in Reference [7]). The present report is divided into two sections which discuss BIT and BIT-related problems in both the concurrent and the nonconcurrent BIT

categories. In the concurrent BIT discussion, exceptions and interrupts are defined in terms that expand upon the explanations found in MIL-STD-1862 and the MCF prime item specifications [1,2]. For example, MIL-STD-1862 does not address the questions of how BIT signals are to be reported to software. Instead, it mentions two BIT signals that make use of the MIL-STD-1862 interrupt mechanism and goes no further. This report analyzes three mechanisms that currently exist in the MCF architecture that could be used fo. reporting BIT errors. Based on this analysis, a reporting mechanism for MCF is then recommended. With the knowledge that many BIT-detected errors will be transient in nature, the idea of a "retry" mechanism is presented, along with an explanation of why it is needed in hardware and why it would be beneficial via software [8]. Finally, an integral method of handling BIT signals using both hardware and software is presented.

The second section of this report discusses nonconcurrent BIT issues and alternatives. An overview of software error handling in the operating system environment is presented. In the context of software error handling, the ability to explicitly test improperly functioning units is addressed along with several instructions that could be used for testing these units. Finally, fault-tolerant software is discussed with particular emphasis on fault recovery.

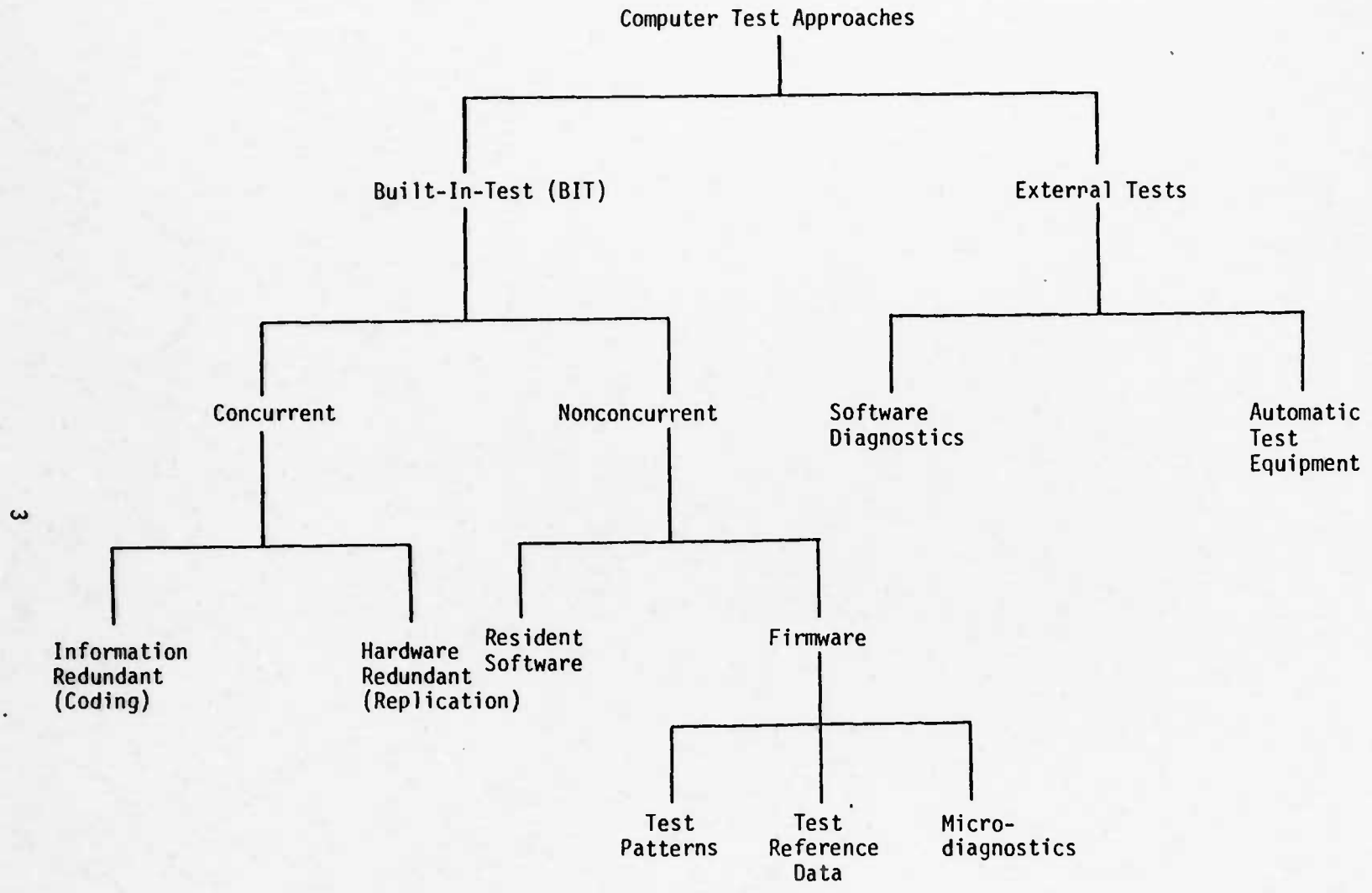


Fig. 1. A Taxonomy of Computer Test Approaches [7].

2. CONCURRENT BUILT-IN-TESTS

Requirements for the AN/UYK-41 and the AN/UYK-49 members of the Military Computer Family (MCF) dictate the incorporation of built-in test (BIT) techniques for fault detection and correction [1,2]. The prime documents for the AN/UYK-41 and 49 specify that "BIT shall eliminate the need for any support equipment to indicate faulty system operation. BIT shall be incorporated to continually monitor system operation." The extent to which BIT is utilized in these computers is not specified and will presumably be left up to the implementation contractor. Conceivably, BIT will vary from implementation to implementation and will be included only to the extent needed to achieve the reliability goals stated which include "upper test" mean time between failures (MTBF) targets between 10,000 and 100,000 hours) [1,2].

Key questions that arise are:

- At what level should BIT-detected errors be handled: user level, operating system level, or hardware level?
- What mechanism should be used to report BIT-detected errors to the software?
- What are the consequences and implications of "instruction retry"?

The following section will address these issues and develop a rationale for their solution.

In regard to the first question, handling all BIT-detected errors at the user level is easily rejected. The user should not be bothered, or even know, that the machine on which he operates is less than perfect. He should not have to write his own BIT handlers; in general he does not have the information for dealing with these errors nor the privilege level required to deal with them. There are numerous other reasons why the user should not be asked to deal with BIT-detected errors. These will not be discussed.

Historically, BIT-detected errors have been handled by both the hardware (instruction retry, error correcting codes) and the operating system (managing bad blocks of memory). Unfortunately, this has not always been a cooperative effort, which has led to the need for the new approaches proposed in this report. There are advantages and disadvantages for BIT-detected error handling in both the hardware and the software. In hardware, the error handler can be designed to be a very specific, selfchecking piece of hardware with limited access by other hardware elements [9]. This reduces the possibility of an error in the handler itself. Software, by contrast, may use the same hardware each time it executes. If the hardware is faulty, then the software execution may fail. Hardware generally implements only one algorithm and may not take advantage of much of the information available to it. Software is much more flexible in that it can realize multiple algorithms based on the information available.

Before discussing the level at which BIT-detected errors should be handled, other issues must be raised. After answering these questions, we will have presented information and ideas that can be used in discussing the "appropriate level for handling BIT-detected errors."

Conceivably, any mechanism for reporting BIT-detected errors to software should not violate the philosophy of the MCF architecture; rather, it should exist within the framework of the architecture and, therefore, be an integral part of it. Three distinct mechanisms for reporting exceptions and interrupts already exist in the proposed MCF architecture. These are the software and hardware exception facilities and the interrupt facility. Each facility is different and each has its own advantages and disadvantages as a means of communicating BIT-detected errors.

The following discussion characterizes MCF exceptions and interrupts and then addresses the reporting mechanisms.

a. Exceptions

There are two distinct types of exceptions, "software exceptions" and "hardware exceptions." The software exception is an event caused by an error in the currently executing software, such as an illegal address, a divide-by-zero,

or a task failure. Because software exceptions have no latency; i.e., they will not disappear with time, they need not be handled at top priority. The event can sometimes be ignored, as with overflow or underflow; in other cases, it must be handled to decide if the program should be aborted, as with invalid access or illegal address. These events should not recur if the handler is "correctly written." The manifestation of the event is selflocalizing.

(1) The Software Exception

A software exception is caused solely by the currently executing program and its data. Such an exception could be repeated by simply re-executing a certain segment of code in a specified environment. It is therefore logical for these exceptions to be handled entirely by the program units in which they occur.

There are three major schemes for coping with software exceptions [10]: signal, notify and escape. All three schemes are essentially similar and differ only on the issue of postexception flow of control. The first two schemes basically allow the program unit in which the exception occurs to resume program control at the point at which the exception occurred. This philosophy is evident in Mesa [11] and Alphard [12]. The third scheme requires local termination (escape) upon detection of errors. This approach has been adopted partially by certain dialects of Bliss (with the SIGNAL_STOP construct) [13] and completely by Ada [14], the latter of which enjoys the distinction of having the MCF architecture as its host machine.

The Steelman [15] requirements for high-order programming languages used by the Department of Defense specify that exception handling shall be of the "escape" variety. This is basically a restrictive approach since it automatically terminates the excepted program unit. It is not, however, nearly so restrictive an approach as it first appears to be, because the caller of the excepted program unit is at liberty to call that unit at a later time, if it determines that conditions are more conducive to its successful elaboration (e.g., pathological data have been eliminated, queueing delays have been

---

In this case "exception" does not necessarily mean "undesired event" but rather "rarely occurring event."

overcome, and timeouts are no longer a hindrance).

Different types of exception mechanisms necessitate fairly different programming styles; hopefully, "escape" or "bail-out" programming is the more manageable and disciplined style of programming. Moreover, termination of the offending program unit is a virtual necessity if formal program verification or correctness techniques are to be employed [14]. This consideration applies to optimization methods as well.

As a real-life paradigm for the type of exception handling discussed above, consider the following procedure:

```
PROCEDURE Gauss IS
  TYPE Vector IS ARRAY (1 .. N) OF Real;
  TYPE Matrix IS ARRAY (1 .. N) OF Vector;
  solution:  Vector;
  M:  Matrix;  -- augmented coefficient matrix
  Singular, IllConditioned:  EXCEPTION;

  PROCEDURE Process [M:  Matrix] IS

    PROCEDURE Pivot [M:  Matrix, n:  Natural] IS
    BEGIN
      -- code to pivot the nth row of M
    END Pivot;

    PROCEDURE Triangularize [M:  Matrix, n:  Natural] IS
    BEGIN
      -- code to lower-triangularize M
      -- arithmentic exceptions may be generated here
    EXCEPTION
      WHEN DivideByZero => RAISE Singular;
      WHEN OverFlow => RAISE IllConditioned
    END Triangularize;

    PROCEDURE BackSubstitute [M:  Matrix]
    RETURN Vector IS
```

```
      BEGIN
        -- solve by substituting values
        -- during the first back-substitution an arithmetic error
        -- may be raised
      EXCEPTION
        WHEN DivideByZero => RAISE Singular
      END BackSubstitute;

    BEGIN -- Process
      FOR n IN 1 ..  N-1 LOOP
        Pivot(M, n);
        Triangularize(M, n);
      END LOOP;
      solution := BackSubstitute(M);
    EXCEPTION
      WHEN Singular => Print("The system has no unique solution");
      WHEN Ill-Conditioned => Print("The system is
        ill-conditioned");
    END Process;

  BEGIN -- Gauss
    more:  String := "Yes";
    WHILE more = "Yes" LOOP
      Print("Enter the augmented coefficient matrix.");
      Read(M);
      Print("More?");
      Read(more);
    END LOOP;
  END Gauss;
```

The Gauss procedure implements the Gaussian elimination algorithm by repeatedly pivoting the rows of the augmented coefficient matrix of a system of N simultaneous linear equations in N unknowns. The Pivot and Triangularize procedures perform the required elementary row operations on the augmented coefficient matrix, and if the divide-by-zero exception is generated, then the program knows that the pivot element must have been equal to zero, in which

case the system is singular. If an overflow exception is generated, then one of the entries of the augmented coefficient matrix is large enough to cause the matrix to have a large condition number (i.e., the system is illconditioned). Exception handling of this sort is not foolproof, nor is it magic, since any number of exceptions may be generated during the execution of a segment of code. However, it does go a long way toward helping a programmer cope with the bizarre, the less-than-mundane, and the novel.

(2) The Hardware Exception

The second exception type is the hardware exception. This event is caused by the hardware. It is not directly related to the software exercising the hardware, e.g., parity, power-fail, BIT. It is characterized by requiring quick handling so as to reduce or limit any data corruption. The event can not be ignored and may recur while executing its handler. The event needs to be localized to the least replaceable unit (LRU) in which the event occurred (not necessarily manifested) for maintenance purposes. It is vitally important that hardware exceptions be tended expeditiously (usually by some specially written trap handler). Barring any further complications, the flow of control should revert back to the point in the program unit at which the exception occurred. This is an explicitly stated, absolute criterion that should be met by the MCF architecture [16] that states, "It must be possible to write a trap handler that is capable of executing a procedure to respond to any trap condition and then resume operation of the program." This presents some special difficulties for architectures (such as the MCF's) that allow instructions to be interrupted in midexecution. It would be desirable to resume the instruction's execution precisely (or as close as possible) where it was cut off. There are critical issues to address with respect to this problem; these are considered below.

These definitions of software exception and hardware exception differ from the definition of exception in the prime item reports and MIL-STD-1862. They are defined in this manner to draw a closer distinction between the MCF idea of exceptions and BIT-type exceptions.

b. Interrupts

Interrupts are asynchronous events generated externally or independently of the executing instruction. They are used to inform the system that some specific action has happened or is about to happen. These events characteristically require rapid handling, due in part to data latency. An interrupt can be ignored if it lacks sufficient priority to receive attention, or it can be deferred until its priority is high enough to insure some attention.

Table 1 lists the characteristics of software exceptions, hardware exceptions, and interrupts. Some of these characteristics are discussed in MIL-STD-1862. Those that are not discussed in the MCF document are discussed in the following section. Some of the other characteristics are also amplified in this section.

c. Exception and Interrupt Facilities

MIL-STD-1862 does not explicitly define an exception. Instead, it says: "Program errors are handled by the exception facility," and "... an exception may be raised by RAISE or ERET instructions, or by the detection of an abnormal condition by the hardware." The phrase, "... or by the detection of an abnormal condition by the hardware," implies that BIT-detected errors are handled by the exception facility. However, some events (such as parity or power-fail) that are defined as interrupts or use the interrupt facility in MIL-STD-1862 clearly fall in the area of hardware exceptions. Based on MIL-STD-1862 as it now stands and prime item reports [1,2,17], BIT exception handling is spread over two separate and distinct facilities. RTI feels that this is not what was intended by the specifications. The MIL-STD-1862 phrase in question could better read, "... or by the detection of range or domain violations by the hardware." This phrase would then specifically refer to signals such as carry, underflow, overflow, truncate or divide-by-zero. In the following paragraphs RTI will discuss why only one dedicated facility should be used for handling BIT detected errors.

Three different mechanisms are used to communicate to exception and interrupt handlers when an interrupt or exception occurs. Exceptions use two of these mechanisms: (1) passing the exception code to the locally defined exception handler or (2) a parameterized call to the supervisor exception handler. The third mechanism is the interrupt and trap facility.

Table 1. Characterization of Software Exceptions, Hardware Exceptions and Interrupts from MIL-STD-1862.

CHARACTERISTIC	SOFTWARE EXCEPTIONS	HARDWARE EXCEPTIONS	INTERRUPTS
Asynchronous or synchronous with respect to program execution	Synchronous	Asynchronous	Asynchronous
Data latency problem	No	No	Yes
Data corruption	No	Yes	No
Need immediate attention	Yes	Yes	Not always
State information need to be saved?	No	Not discussed	Not discussed
Priorities required?	No	Yes	Yes
Instruction retry	No	No	Yes
Maskable	Yes	No	Yes

(1) Procedure-associated Exception Facility

Local exception handlers are segments of code within a procedure with which the handler is associated. The actual association is made at procedure entry by setting the "exception handler specified" bit of the entry header or by executing an EXCEPT instruction which provides the address of a code segment which is invoked if an exception is raised. These handlers are not procedures but segments of code to which the program branches if an exception is signalled.

The only information available to these local handlers about the exception is the exception code generated by the exception. There is no capability nor information that allows these handlers to return to the location at which the exception is raised; thus local exceptions are terminal exceptions for the procedure with which they are associated.

(2) Supervisor Exception Facility

The other exception handler is the supervisor exception handler, which is permanently associated with every task. Whether the supervisor exception or local exception handler is invoked is determined by the up/down level exception (UDLE) bit in the processor status word (PSW) of the machine. This handler is invoked like a procedure call and thus has its own execution frame. It is also invoked as a privileged task on the kernel context stack. The information passed to the supervisor exception handler is the exception code, the address at the beginning of the instruction that was executing, and the program counter of the context which invoked the supervisor exception handler.

The supervisor exception handler was designed to work in the debugging environment, not to be a part of the debugged system [18]. The only other time the supervisor exception handler is invoked is when an exception has propagated to the base of an execution frame. It is then invoked with a task failure exception, not with the exception that was raised in the base context. In this case it is invoked with the task failure exception to act as a buffer between the task with a task failure and the task that spawned the failed task. In this way no exception is passed to the spawner. Recall that in the MCF architecture exceptions handled by the local handler

eventually cause the termination of the context in which they are raised. So the spawner's context that handled this exception would have to terminate, were it not for the buffer zone provided by the supervisor exception handler.

(3) Interrupt Facility

The interrupt and trap facility is treated as a parameterized call with the address of the handler held in a vector. The parameters for each entry are defined by the MCF architecture. The vector and implicit priority are also defined by the MCF architecture. When an interrupt occurs, the priority of the interrupt is checked against the priority of the executing task. If the interrupt's priority is higher than the executing task's priority, the interrupt takes effect immediately. A new context is built with the address in the vector location used as the address of the interrupt handler's entry point.

The interrupt facility has several advantages as a BIT-detected error handler. The correct handler is immediately invoked upon receipt of a BIT-detected error. The necessary information can be passed as parameters to the handler routine, and these need only be defined in the architecture. BIT-detected errors can be grouped according to levels of severity, with the most severe errors invoking a hardware routine similar to power-fail to save the status of the machine. The handler is a procedure; thus it can be exited, and, with proper programming, the instruction where the error occurred can be resumed or restarted.

Comparing the needs of the hardware exception handler with the characteristics of the interrupt facility, one can see that they fairly well match each other in terms of needs and abilities.

The MCF architecture currently recognizes that parity errors and power-fail are best serviced by the interrupt facility. It should be obvious from the above discussion that the remaining hardware exceptions should use the same mechanism.

d. Retry

The third question regarding concurrent built-in-tests concerns the implications of "instruction retry." The capability to retry an instruction (this includes resuming an instruction) has historically been a hardware capability invisible to the programmer. This section discusses the problems and differences of retry after hardware exceptions and interrupts. This discussion is basically the hardware view of instruction retry. The final subsection discusses different instructions that can be used to explicitly control retry from the software level. This approach is based on the idea that if a BIT-detected error is handled in software and results in the operation being corrected, then a posible alternative is to retry the "interrupted instruction."

"Interrupted instruction" will be used a great deal in the following paragraphs. It should be understood to include the occurrence of an interrupt and also the occurrence of a hardware exception. It does not include software exceptions.

(1) Returning from Hardware Exceptions

Many BIT-detected errors are manifestations of transient faults. As circuit density increases, the percentage of transient faults to overall faults will increase, e.g., as a result of substrate-generated alpha particles. Because so many of these errors are due to transients, the idea of a retry capability appears attractive [19, 20, 21].

In an earlier MCF report [16], the retry capability was listed as a desirable asset. In the MCF architecture, the appropriate granularity for specifying retry is at the instruction level. To specify retry at the instruction level, the address of the beginning of the interrupted instruction must be available.

There is a problem with this approach -- what if the BIT-detected error occurs while in the instruction execution cycle? Retrying the instruction could produce erroneous results if some information in program visible storage were altered while executing, e.g., if a partial block move or edit were performed. Since the ability to resume an instruction is desired, all possible BIT-detected

errors need to have manifested themselves before a change is made to the program visible storage in each iteration of the execution cycle. If this is possible, the machine can resume execution at the point following the last change to the program visible storage, assuming that the state at that point is saved or can be reconstructed [18]. As an aid to understanding the following paragraphs, a model of the instruction cycle is provided in Figure 2. An example to illustrate the concepts of "retrying" and "resuming" follows.

In the figure, an instruction, move block (MOVBLK), is being executed. The machine is in the operand evaluation cycle and a BIT-detected error occurs. Based on information available about the state of the interrupted instruction, the handler decides to retry this instruction. The handler issues a RETRY instruction and supplies the address of the instruction to be retried. Since instruction fetch and operand evaluation have no side effects other than incrementing the program counter (PC), RETRY can be done without saving any more information than the address of the beginning of the current instruction. This value is defined in the ISP of the MCF architecture as old.PC, see Appendix A, and it is passed as a parameter to the supervisor exception handler. This is to indicate the availability of this datum in the current specification of the MCF architecture.

In an instruction such as the MOVBLK the microcode that implements the execution cycle is generally executed as a loop, with the loop control as the count parameter of the instruction and the body of the loop as an execution phase. To RESUME an execution cycle, the internal state of the machine must be preserved or reconstructed every time the execution phase loop is executed. This internal state can include the opcode (points to the microcode), current address of the source and destination, and the current count of the loop. The information saved and the amount saved is necessarily implementation dependent. A discussion of where this information could be saved will be deferred until later.

(2) Returning from Interrupts

Another issue that pertains to the above discussion of retry and resume is the interruptability of instructions. Several instructions are stated to be

Instruction:
MOVBLK #5,Source,Destination

CYCLE	ACTION
Opcode Fetch	IR<-MOVBLK
Operand Evaluation	
Phase 1	OP1<-#5
Phase 2	OP2<-Source
Phase 3	OP3<-Destination
Execution	
Phase 1	Source-->Destination
Phase 2	Source+1-->Destination+1
Phase 3	Source+2-->Destination+2
Phase 4	Source+3-->Destination+3
Phase 5	Source+4-->Destination+4

Fig. 2. Cycles and Phases in Instruction Interpretation.

interruptable (string instructions). If they are interruptable, how are they restarted? Again, this is the idea of resuming an instruction. The explicit specification that an instruction is interruptable implies that the instruction cycle is interruptable and thus can be retried or resumed. Is the capability to retry or resume an explicit machine instruction fully understood by the programmer as to its requirements, drawbacks, and side effects, or is the capability "hidden" from the programmer in such a way that he does not know that he is returning to an instruction that will be resumed or retried? MIL-STD-1862 does not answer any of these questions explicitly, so it is a fair assumption that the capability to resume, at least, is a hidden capability.

What does this mean? It should not be concluded that interrupts and BIT-detected errors are one and the same. A BIT detected error means that something incorrect has happened and if the current instruction runs to completion a possibility exists that incorrect data will be stored in the program visible storage. On the other hand, an interrupt is an event that says something needs attention, but can way wait for the current instruction to go to completion. Some instructions are interruptable because they can operate on as many as 2**32 bytes of memory with one instruction. To complete the distinction between these two items, an interruptable instruction is one in which the execution cycle can be suspended once a certain "point" (the end of a phase) in the instruction cycle is reached. The internal state saved is the state after completing the current phase of the execution cycle. A BIT-detected error says: "Stop what you are currently doing and save as internal state the internal state as of the last change to program visible storage." Internal state refers to memory elements in the machine which are not programmer visible but define the status of the machine at any phase in the instruction cycle. If the machine is in the execution cycle, phase N, an interrupt will complete the execution of N and save the state in order to resume in phase N+1, while a BIT-detected error will abort phase N and save the state of phase N in order to to restart phase N, as shown in Figure 3.

(3) Returning from Software Exceptions

The software exception is the third type of undesired event that may

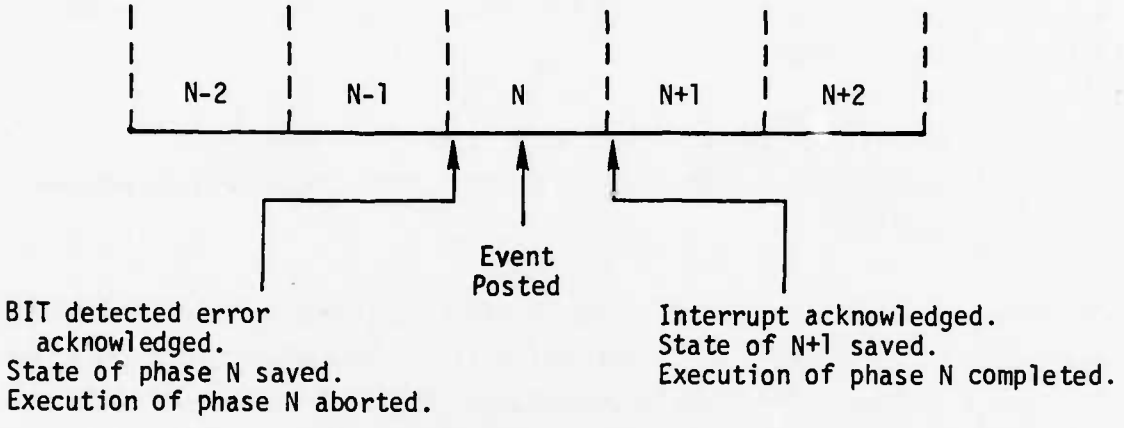


Fig. 3(a). Multiphase Execution Cycle.

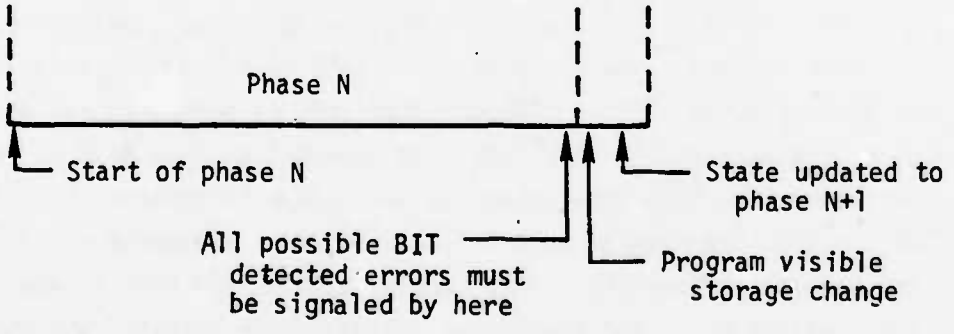


Fig. 3(b). Actions in a Phase.

impinge directly on instruction resumption. The MCF instruction set is a sophisticated procedure-based language with exception handling capabilities. As such, we must carefully consider the semantics of program resumption, abortion, and retry after the handling of BIT detected errors. The two major questions in software exception handling are:

1) Who handles the exception once it has been generated?
2) What happens to the overall program after the exception has been handled?

The first question engenders a number of other questions: should the handler be statically or dynamically specified, where is the exception handler specified, is there a hierarchy for handling exceptions, etc.? To answer the second major question, one must first consider how the program returns from the exception handler. Does the excepted program unit terminate, skip, resume, or retry, etc.? Furthermore, one must consider how these problems can be solved by the MCF architecture.

"Exception" will generally be used to refer to the detection of a condition which merits special attention. The terms "exception," "condition," and "signal" are often used synonomously. Software exceptions may occur at any point in the execution of a program; they may be defined by the programmer or predefined by the system (e.g., overflow conditions, divide-by-zero, etc.). When an exception occurs within a program unit (e.g., procedure, block, loop), the unit is said to be excepted or is said to signal the exception (presumably to other program units). The excepted unit is sometimes called the offending unit or the signaler. Sometimes the excepted unit is said to raise the exception, but here terminology is not always consistent (cf. [11], [6], [22]). At the time the program unit is excepted, some handler must take control (i.e., the handler is invoked). An exception handler can best be described as instructions which the programmer intends to be executed whenever a certain exception is signaled. A handler is simply a sequence of instruction statements, but different programming languages (if the language implements exception handling) have different syntactical and semantic rules for their

handlers. For instance, one language may think of an exception handler as a statically defined trap routine which is executed and returned from when the exception is signaled, while another language associates an exception handler with a procedure-like entity which is dynamically determined at run time. Once the handler is invoked by the exception and its context (e.g., the excepted program unit), it is executed as a normal sequence of instruction statements. After the handler's execution is completed, control will return to some point determined by the semantics of the language's exception handling facility. In the general model of exception handling it is assumed that a procedure-based language is used. "Procedure" is used to mean program text that can be activated by reference to the unique name associated with the procedure. Thus machine language is discussed as well as high-order language. A procedure P is activated by procedure Q's call to P. (A procedure is allowed to call itself.) The call relationship which exists at a given point in a program's execution implicitly defines the program's activation record or procedure stack. Disciplined programming practices require that each procedure be seen as the implementation of an abstraction (e.g., a mathematical function), preferably with associated documentation or functional specification of the implemented abstraction. An important principle in procedure-based languages is the principle of information hiding: a program calling another program only requires knowlege of the callee's abstraction (essentially its input/output relationship) and needs to know nothing of the callee's implementation details.

Now that the basic exception handling model has been addressed, the previously raised question of who shall handle a signaled exception can be discussed. First, it must be clear how we specify an exception handler. There are three ways to do so:

(1) The handler can be specified as completely static: each time the exception E is signaled, the handler H(E) is executed. This is equivalent to writing a trap handler for a condition.

(2) A handler is associated with each procedure-exception pair [6]. Here the occurence of a given exception E in procedure P can be handled by H(E,P), but the occurence of E in procedure Q might be handled entirely differently by handler H(E,Q).

(3) The second alternative may be extended by allowing a procedure call (not just the procedure body) to have an exception handler associated with it. Thus, if P1,...,Pn are the various calls made to procedure P in a program, handlers H(E,P1),...,H(E,Pn) may be available to handle E.

Throughout the above discussion we are assuming that we are limited to binding the various handlers for a specific exception to elements of the program's activation record. Reference [12] shows such an assumption to be unnecessary, but we will restrict ourselves to this nonetheless. In general, static exception handlers have limited power, so this report will consider exception handling models that employ a dynamic binding of the exception handler with a combination of the exception and the exception's context.

Given that an exception handler for exception E is program text which is somehow associated with the various procedures of a program, there is still the problem of how to decide which of the handlers for E (and in general there may be several) will be initiated when an excepted procedure signals E. The procedure whose associated handler handles a signaled exception will be called the catcher of the exception.

First, let us review some of the current methods actual programming languages use to bind exception handlers to procedures. The MCF architecture specifies that every procedure has the option of specifying an exception handler. The same is also true of the language Ada [14]. In both cases the exception is program text which is appended to the end of the procedure's body. Generally, when an exception E is signaled in procedure P, the associated handler H(P) examines the value E and, depending on the value, transfers control to the appropriate section of H(P) (thus one writes H(P,E)). The idea in both the MCF architecture and Ada is that when procedure P signals exception E, control is diverted immediately to the handler H(P,E); in other words, the catcher and the signaler are the same.

The CLU language also associates handlers with procedures by including the handler at the end of the procedure body. Unlike the MCF architecture and Ada, however, the catcher is the procedure which called the signaler. So, if procedure P calls procedure Q which later signals exception E, the resulting action transfers control to H(P,E), the handler for E in P.

Mesa is by far the most liberal language with respect to exception handlers. Mesa's handlers may be associated either with a procedure (the handler is included at the start of the procedure body by the ENABLE clause) or with a specific call to a procedure (via a catch phrase). Conventional scope rules determine which handler is employed (e.g., an ENABLEd handler takes precedence over one specified by a catch phrase). In the Mesa scheme, if procedure P signals exception E, either P's handler for E or P's call's handler for E assumes control.

In all of these languages except CLU, the signaled exception may not be caught by the caller of the signaler, in which case the signal is simply propagated further up the call stack. Appealing to the principle that only the caller of a signaler should know that the signaler has signaled an exception, CLU has taken the unique position that an exception may be propagated from signaler to caller, but no further. Otherwise, if a procedure handled exceptions that originated deep in the bowels of other procedures it had called, this would imply a knowledge of the implementation of the callees on the part of the caller. Given all these *different mechanisms* for defining exception handlers, it may be advisable to have an architecture which is adaptable to these various mechanisms. With minor modification the MCF architecture can directly support the binding and control transfer mechanisms discussed above.

The question of where to go after the exception has been handled is still open. The MCF architecture, Ada, and CLU all agree that after the exception handler has executed its last instruction statement, the signaling procedure must terminate. The program then resumes at the point following the call to the signaler. This approach is taken on the grounds that the called procedure should not depend on the actions of its caller, once the call has been initiated -- the called signaler's resuming after the calling catcher's handling of the exception would violate this. Mesa allows the signaler to be terminated, to RETRY the signaler by recalling it, to RESUME at the point in the signaler where the execution was signaled or to CONTINUE at the instruction following the signal. Whether the last mechanism can be implemented directly in the MCF architecture without significant modifications is not clear. However, it could be supported by the MCF architecture at the cost of additional overhead (e.g.,

by including a runtime procedure Signaler which passes the signal to each handler in its turn).

(4) Saving the Interrupted State

Retrying an instruction from hardware or software requires that a certain amount of state information be saved on acknowledgement of an interrupt/BIT- detected error [18]. How much information needs to be saved is necessarily implementation dependent? A method for determining if the previous context was in a "retryable" state is presented.

There are currently two bits available in the PSW where state information can be stored, bits 2 and 3. If this is the maximum number of bits available, only four states can be encoded. How is the instruction cycle broken up into representable segments by these two bits? Basically, resuming an instruction can occur anywhere in the instruction cycle as long as sufficient information is saved which can represent the "point" in the instruction cycle uniquely. Depending on the "point" this can be an excessive amount of information to save. Logical points, where an instruction can be interrupted, are after each operand is evaluated or after each execution phase. For interrupts this entails completing the operand evaluation or execution phase and saving the state beginning at the next "logical point." For BIT-detected errors the current activity is aborted and the state variables for this control point are saved. In this report we have mainly referred to resuming if the event occurred in the execution cycle, because program visible storage may already have been changed in prior execution phases. Resuming after an event can easily encompass operand evaluation. However, throughout this report we will assume that one resumes from an event in the execution cycle or retries an event in the instruction fetch or operand evaluation cycle. An example encoding of the PSW bits follow:

00 - indicates a "normal" state. This means that the "calling" procedure was not "interrupted" while in an instruction cycle.
01 - indicates a "retryable" state. The procedure was "interrupted" while in either the instruction fetch or the operand evaluation cycle.

10 - indicates a "resumable" state. The procedure was "interrupted" while in the execution cycle of an instruction and the "internal state" of the machine at that time is saved.
11 - reserved, could also indicate that an incomplete save was done, and so resumption would be dangerous.

These bits would be set by an interrupt signal or BIT signal.

The current thinking is that the "internal state" and the state of the interrupted instruction should be saved in the context of the handler. It is conceptually cleaner to save this information in the context of the interrupted instruction but on the surface it appears to be unwieldy for the hardware. Why? Any instructions in the handler that may wish to interrogate the status bits or the internal state could only do so with great difficulty. If the internal state were saved in the interrupted context, either a separate piece of "hardware" or a revamping of the current hardware would be necessary to remove this internal state and keep the context pointer in order.

(5) Software Retry

This section discusses three approaches to explicitly control resuming or retrying an instruction from the instruction level. If a BIT error handler is to identify an error, isolate the faulty module, and continue computing, there must be some way for the error handler to allow computing to resume at the point where it was interrupted. By the same token, the same capability to resume computing is necessary to return from an interrupt handler.

Let us consider the capability to explicitly retry or resume an instruction. RETRY requires the address of the interrupted instruction. The address can be passed as a parameter at handler invocation time. The instruction should be privileged, and the address should not be explicitly stated as an operand. Instead, it should be an implied operand, such as

"normal," CONTINUE operates like the RETURN. The difference between this approach and the second approach is that CONTINUE can be made privileged, which removes it from the purview of the nonprivileged user. There would also be no way in which the state bits could be modified from "resumable" to "retryable".

There are a few more instructions that could be useful in supplementing the previous approaches. In one of these the state of the interrupted context is tested. This instruction would then set the condition code bits. For example, a "normal" state could clear all the bits, a "retryable" state could set the "N" bit, and a "resumable" state could set the "Z" bit.

Wuerges and Parnas [23] have advocated three instructions for use in undesired event handling. Undesired events map into our classification of "hardware" and "software" exceptions and interrupts. Two of their proposed instructions, RETRY and CONTINUE, are basically equivalent to RETRY and RESUME. Wuerges' third instruction, CLEAR, ignores the "interrupted instruction" and starts interpreting the next instruction in sequence. The instruction, more importantly, resets the program visible memory to its value at the beginning of the "interrupted instruction." This is impractical if a CLEAR is executed on an interrupted MOVBLK instruction which was halfway through moving a page of memory. In its place we would propose an ABORT instruction which would discontinue the interpreting of the "interrupted instruction" and would begin interpreting the following instruction. But, the program visible memory would not be reset. The programmer should be aware that ABORTing in the "resumable" state does not undo the changes of the earlier execution phases in the execution cycle.

e. Level For Action

The previous sections discussed why a retry capability is needed in the MCF architecture, and presented a brief explanation of hardware and software retry. Regarding which level--hardware or software--is better for retry, RTI proposes a combination of the two levels in order to take full advantage of the inherent strengths of each level. Presented below is a scenario that incorporates hardware and software in a integrated system for dealing with BIT-detected errors.

Upon receipt of a BIT-detected error, the hardware saves the machine's state variables and attempts a retry. If it succeeds, the error is considered a "soft error" and processing continues in the normal fashion. If the hardware retry fails, it tries again several more times. If all attempts fail, the BIT-detected error is considered a "hard error" and the error is passed to the software handler. At this point software, in the guise of the error handler, is invoked and not before. If the software can "fix" the problem it can retry the "interrupted instruction" and continue task execution. Figure 4 presents a block diagram describing the actions taken in the event of a BIT-detected error.

Instruction retry is a hardware capability which is shared at the instruction level with the programmer via some of the previously mentioned instructions. While the capability to retry exists at the instruction level, it can not be properly invoked if the state of the previous (interrupted) context does not reflect a retryable state. This state can only be set by the hardware when it raises a BIT-detected error or an interrupt is received. The state indicator can only be cleared by the hardware when it executes one of the retry-type instructions. The handler is invoked in the same manner as a procedure, but the parameters saved in its context are specified by the architecture. The state variables of the interrupted context are saved in some appropriate fashion so that they can be restored.

The software handler can implement several algorithms and use the appropriate algorithm, depending upon the information it garnered while analyzing the machine. Upon receipt of a BIT-detected error the hardware does an automatic retry, if this fails, it can continue issuing retries for a specified number of times. Because transient errors often appear in bursts [24] it may take multiple attempts before the transients disappear. Ng and Avizienis [25] suggest that the hardware scheme have some built-in delays in order to "wait out" the error burst. Ng and Avizienis [25], Sedmak [19] and Carter [26] also strongly suggest a multilevel recovery strategy that involves more than just an instruction retry. Every retry attempt is automatically logged. If the hardware retry fails, the appropriate software handler is invoked based on the "error code/address." In the case of a severe error, on the same level of magnitude with a power fail, the hardware could do a series of retries; if that failed, the hardware would try to gracefully close down.

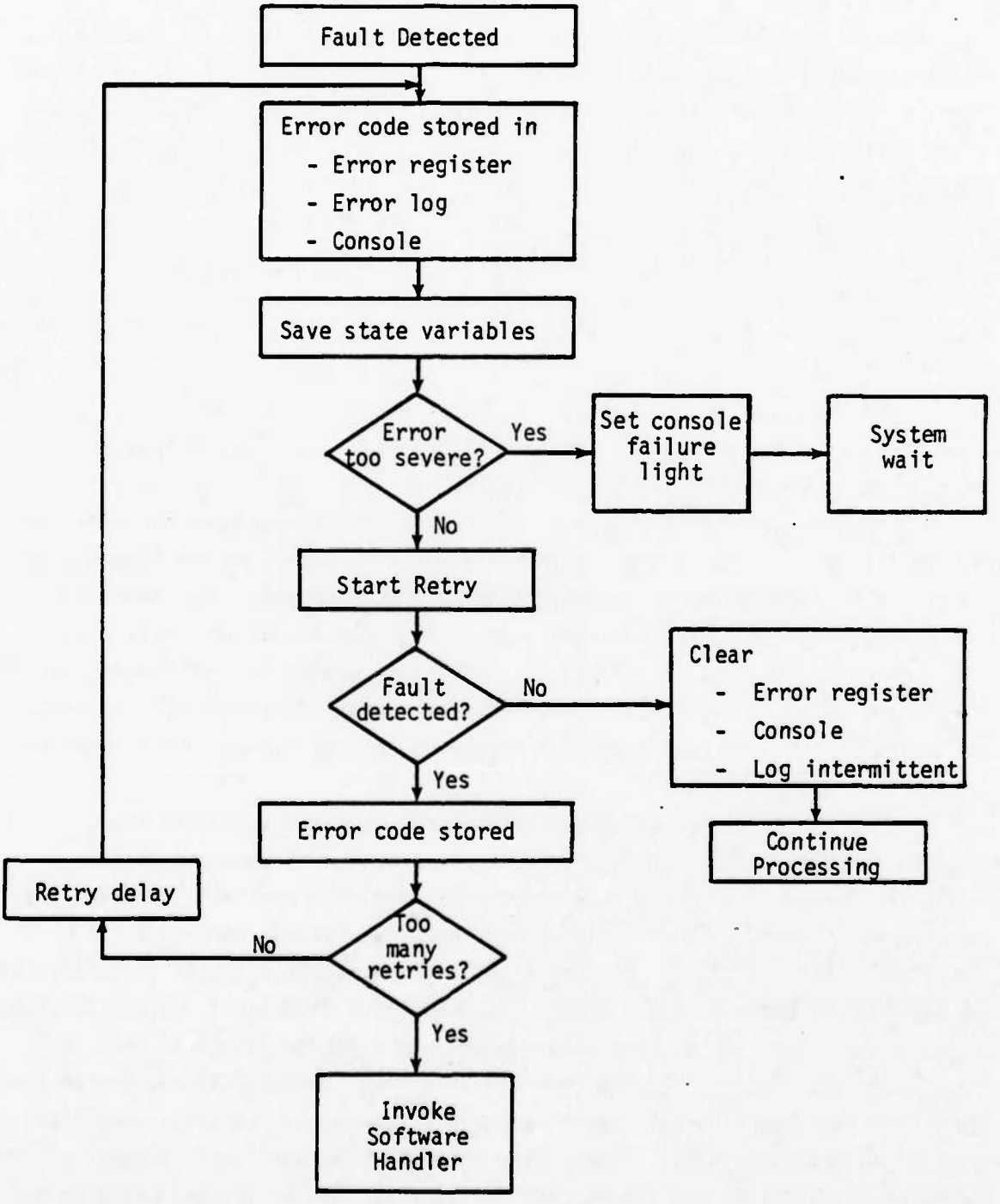


Fig. 4. Recovery Block Diagram.

The point can be raised that the hardware part of the process could be done entirely in software, but there are many important advantages to allowing a purely hardware approach to the problem. For example, the extra control overhead needed to do the series of retries in hardware is a small fraction of the retry hardware that will be required. The hardware can be a separate piece of selfchecking hardware that can be isolated to a large extent from the rest of the hardware [9]. The time, difficulty, and space requirements required by a purely software approach is very large compared to the small hardware overhead required.

This proposed recovery process is a total package which makes efficient use of the different strengths at each level. An extension of this strategy to yet another level will be discussed in the following section under fault tolerant software. This is the idea of doing rollback-and-recovery. Interrupt recovery is easily accommodated into this recovery process by simply bypassing the hardware level retry efforts.

3. NONCONCURRENT BUILT-IN-TESTS

Whereas concurrent BIT is predominantly implemented in hardware, nonconcurrent BIT relies almost entirely on software and firmware for its implementation. As in any well structured program nonconcurrent, software-based BIT is useful because it is modular, portable, modifiable, maintainable, easy to understand, and properly designed for human interface. The inherent weakness of software-based BIT, of course, is that it depends on the very medium which it intends to test. However, experience has shown that intelligently designed software-based BIT can be invaluable to the success of a computing system design.

Consider a computing system based on the Indy 500 principle. In this scenario the overall computing system is comparable to the race. The operating system includes the operating system and applications programs, the architecture, and the machine implementation. The driver is the low level monitor of his race car's health and performance; he exercises a great deal of control over the decisions which govern how the race is to be run. He is directly in touch with his machine via the instrument panel and the feel of the car, and the decision to continue a lap or pull into the pits when the oil pressure is abnormally high is entirely his. Just as the driver presumably has the ability to make the correct judgement in matters concerning his racecar, there is a sound strategy or algorithm the computing system can use to monitor system behavior and act appropriately. One simplistic algorithm is to abort whenever a malfunction occurs, which is analagous to stopping the car and being towed into the pits whenever the engine temperature exceeds the limit. Other more sophisticated and practical techniques are clearly possible. In the event of any anomaly, the driver should be able to complete the race or at least get his car into the pits. This is the least to be expected of the low-level, first echelon BIT (i.e., concurrent BIT). It is generally catastrophic if a program is oblivious of recent hardware faults and continuous execution -- if the driver is aware of a malfunction, but continues the race and the engine blows at

170 mph -- the driver kills himself, four other drivers, the NBC camera crew, and eleven spectators. Having coasted into the pits, the driver can give control of the car to the pit crew, who can quickly diagnose and repair the faulty car. Just as the pit crew is amazingly efficient, so is the software-based BIT. Sometimes pit crews can not make the necessary repairs, either because the malfunction is serious or there is insufficient time. In this case the racing team has to throw in the towel. By the same token, software diagnostics will occasionally have to simply terminate and signal that external tests and repair are necessary to revive the system.

Once an executing program has been interrupted by some undesired event, what happens? As previously discussed, an interrupt-like signal is generated and termination is suspended. Depending on the information communicated by the interrupt, vectoring to some location occurs and a handler is invoked. The handler is conceived to be a system-level program which runs on the kernel context stack and enjoys certain powerful privileges.

The handler should be part of a larger diagnostic task. This diagnostic task could be broken down into a system exerciser (SysEx) and system files for the purpose of error logging. The SysEx is logically composed of and exercises control over subroutines which would exercise specific modules of the system (e.g., memory, CPU, ALU). These module exercisers could of course be further decomposed into submodule exercisers which would target specific subsets of the modules components (e.g., relatively device-dependent items such as boards, ICs, or register sets). The diagnostic error logs are vital records of the system's behavior. They might comprise a pseudo data base which could be written by the SysEx whenever errors occurred. Organization of the error logs by attributes such as module of occurrence, date of occurrence, frequency of occurrence, and threshholds for errors would be a highly desirable feature. Thus the SysEx could consult the error logs and base its decisions on information provided by the logs [27].

Any handler would be invoked with enough information to enable it to begin locating and containing the fault. This information might be as specific as the address of a failing byte, or as simple as a message that something is amiss.

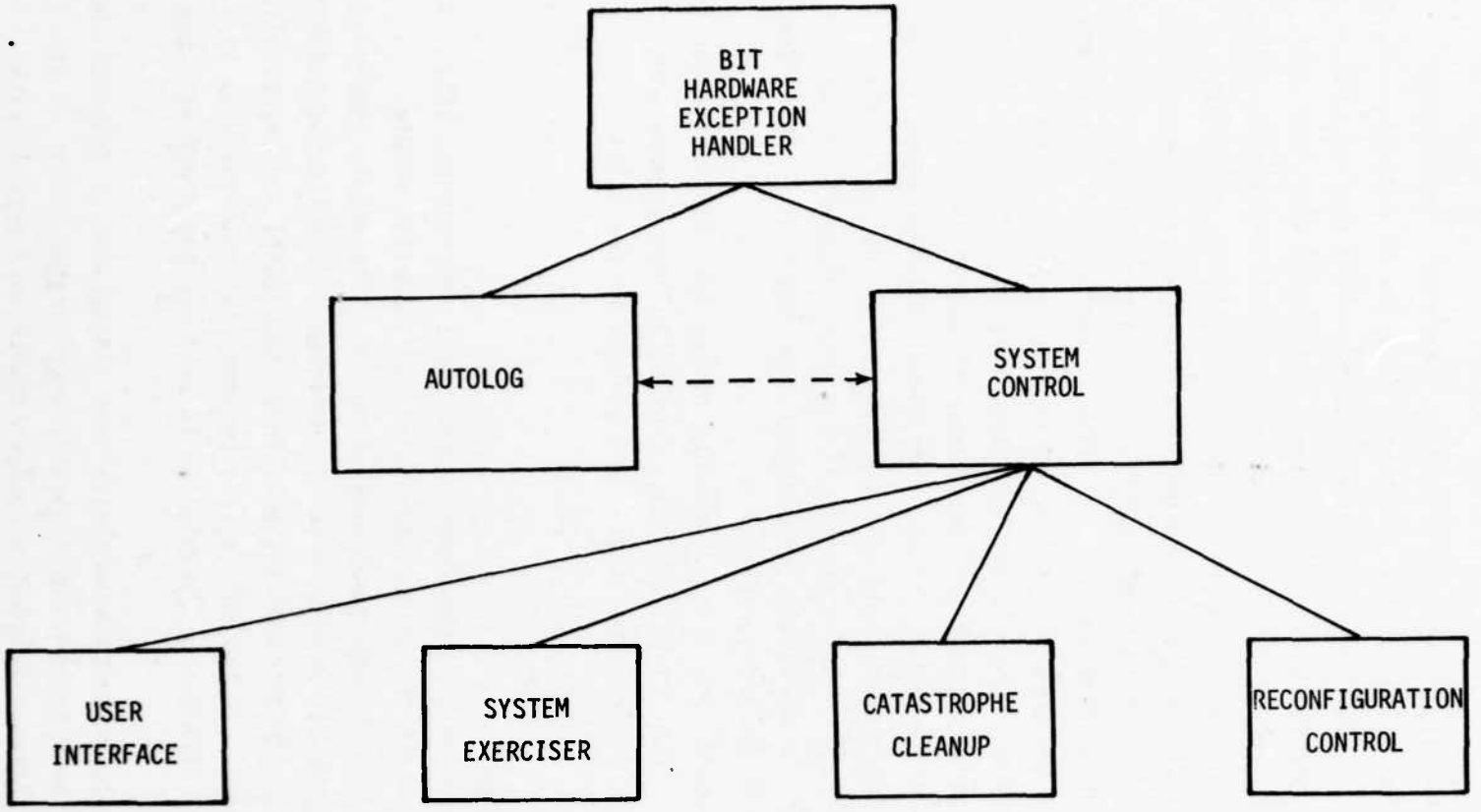


Fig. 5. Top Level Organization for Software-Based Diagnostics.

There would also be a priority associated with each interrupt. The handler would use the passed information and the associated priority to determine how to localize the fault. The handler would in turn call the module exerciser(s) which it has decided is appropriate. It would pass the module exerciser any information it considered relevant. The module and submodule exercisers might test their corresponding hardware components indefinitely (i.e., terminated by some external condition), for a specific number of repetitions, or just once. These exercisers would undoubtedly check to see that data paths were open, verify that the component's input/output relationships remained invariant, and examine the integrity of the component's associated test patterns. The module exercisers could then return values to indicate the state of their associated hardware modules. The handler could call any number of module exercisers any number of times. Results of these tests would be simultaneously logged in the error logs. The operating system would have the results of the SysEx made available to it, thus offering it the opportunity to avoid usage of faulty modules and/or use surrogate modules as replacements for the faulty ones after notifying the user of this reconfiguration.

The SysEx concept would require the expansion of the MCF instruction set to accomodate some specific instructions for testing. The following paragraphs discuss several different instructions that could be used by the SysEx.

a. Test Instructions

The first instruction is a simple, module-level test instruction, TEST "module#," where "module#" is the unique address of some specific module. Module here can mean an LRU or some subelement of an LRU. The module number is the same number returned by BIT when an error is detected. When TEST is issued a set of test patterns are "read" into the unit under test (UUT) and the results are compared to a standard. If the UUT fails a pattern, it indicates this in some manner. Of course, BIT must be disabled so it will not interfere with the testing.

The following example presents a mechanism that can be used to indicate the success or failure of a test pattern and logically incorporates a test of BIT for false alarms. Each pattern is given a number "i"; if the pattern fails, (does not correctly compare), bit "i" is set in a syndrome register, the contents of which are automatically logged at the conclusion of the instruction.

The output of the comparer and the output of BIT are ANDed together to indicate whether the test pattern caused a failure (see Figure 6).

Comparer	BIT		Syndrome
PASS	PASS		PASS
PASS	FAIL		FAIL
FAIL	PASS		FAIL
FAIL	FAIL		FAIL

Comparer AND BIT = Syndrome

In a better method, a bit is set in a BIT syndrome register and BIT sets bit "i", depending on its state, after executing each test pattern "i". This information is then used with the syndrome register to indicate whether a "false alarm" has been raised, if the comparer is in error, or if the indicated test pattern passed/failed (see Figure 7). This added information does require more decision making. For instance, if the comparer and BIT disagree, who is in error? This quandary can be reduced somewhat by making the test circuitry hardcore using self-testing methods [9]. Any errors in this mechanism which are detected by the self-testing mechanism would raise a high priority BIT error.

The second test instruction works like the first, except it is based on the premise that the machine is implemented as a series of concentric layers surrounding a core. This is the same idea as a protected kernel of an operating system. The core of the machine can be implemented in hardware and the other layers can be implemented in software. As the need for speed increases, the software layers can be replaced by hardware versions, until the whole system is implemented in hardware.

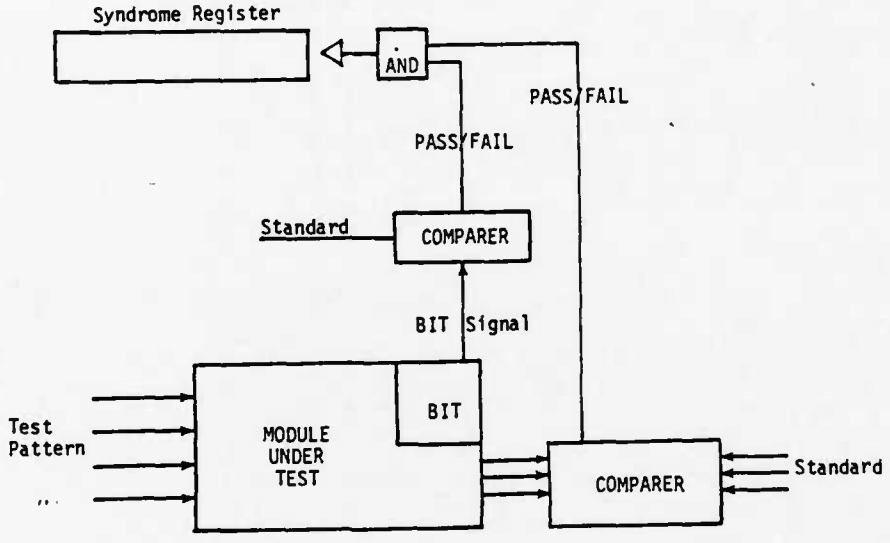


Fig. 6. Syndrome Register for Module.

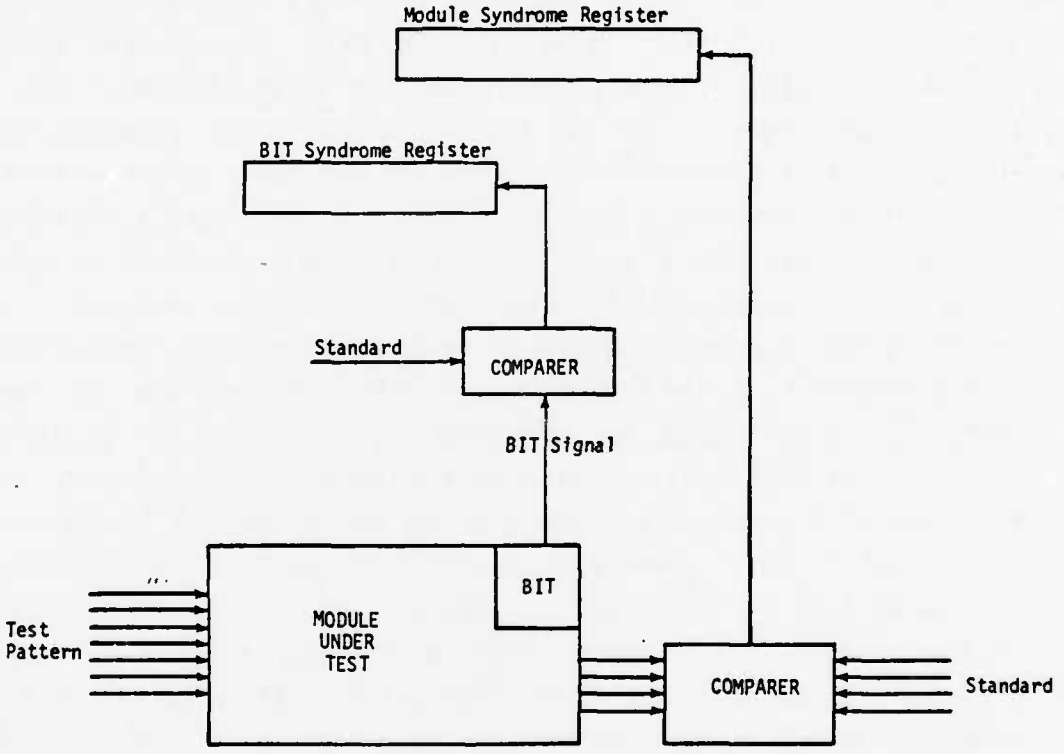


Fig. 7. Syndrome Registers for BIT and Module.

This approach could be extended all the way to microcoding parts of the operating system. Each concentric layer j requires the innermost concentric layers, 0 to j-1, to be correctly functioning before it can function. Checks are performed on all information leaving or entering a concentric layer (this is the idea of "mutual suspicion" in software [28, 29]). The BIT error number represents the concentric layer at which the error was manifested. The advantage of this scheme is that all functions based in the concentric layer j and greater would be considered in error if BIT returned j as the error number. The test program invoked by a BIT signal could then test layer j and beyond, knowing that the test program was written using functions defined in layers 0 through j-1. If the machine's functions were cross-referenced aganist the concentric layers, then all functions in these layers could be marked "disabled" -- DISABLE "layer#." In the case of instructions, one could then make use of the OPEX facility and execute any "disabled" instructions by their software equivalent. (The OPEX facility (unimplemented opcodes) is a vectoring facility which is used if opcodes are executed that are not implemented in the machine.) When the layer in error is replaced, the machine could "re-enable," CLEAR "layer#" as well as the "disabled" instructions, and then continue processing. One of the original ideas behind the OPEX facility was to permit an implementation of a minimal subset of the instruction set in hardware and use OPEX to trap unimplemented opcodes to software versions, thus reducing the microcode of the implementation considerably. This could well be used for a set of decimal-oriented instructions that are infrequently used on a more scientifically oriented implementation.

In nonconcentric machines a module can be used for various functions. If the module fails and a BIT signal is raised, it is possible for the program invoked by the BIT signal to use a function that requires this module. The use of a concentric machine is in itself a means to help isolate errors and prevent the propagation of corrupted data to other sections of the machine. The nonconcentric machine can also take advantage of the OPEX facility if the faulty function can be effectively isolated from further use. A cross-reference table of modules against functions would be required. This table could be very

complex and unwieldy if the implementation is not carefully thought through with error isolation in mind. DISABLE "module#" would disable all functions that make use of "module#" and CLEAR "module#," would re-enable these functions.

Other instructions for testing the machine would be specific instructions that allow one to inspect and modify the internal registers of the machine. These would be necessary, in any case, for the maintenance personnel. An instruction that could be used at the instruction level for checking the integrity of a transmitted byte stream is a cyclic redundancy check, which employs a check polynomial up to 32 terms [30].

The upshot of the above is that the MCF architecture should directly support, via its instruction set, a host of well-known fault tolerance and self checking techniques, e.g., module isolation, memory parity codes, concentric layering, and cyclic coding for data transmission.

b. Diagnostics

A particularly important aspect of nonconcurrent BIT is programmed diagnostics. Programmed diagnostics are an especially flexible and inexpensive way of insuring the ultimate maintainability and reliability of a system. Programmed diagnostics have other advantages as well. They can be run more quickly than external or manual tests. They are less likely to return erroneous diagnoses than human testers. Assuming that support hardware and software are functional, one can trust programmed diagnostics to always execute the complete set of diagnostic procedures (often highly complex) needed to check out an observed or hypothesized failure.

Once the first-echelon BIT facilities have detected and reported an error condition, it may be necessary to execute the resident diagnostic routines in order to localize the error. Diagnostics are also of value as a self test at startup or powerup. When a powerup signal is generated an abbreviated diagnostic routine may be called to perform a low-level checkout of the system's major components. This includes such actions as verifying that all circuitry is functional, writing and reading certain known patterns (e.g., all ones, all zeros, alternating ones and zeros) in all memory locations and registers,

transferring such patterns over the busses with checks of the data at source and destination, noting that all devices respond correctly to wakeup requests, or executing short segments of code which produce known outputs and checking those outputs. The principal uses of diagnostics, therefore, will be for verifying the soundness of the system in initial startup tests; precautionary diagnostics while the system or module is idle or dormant; diagnosing modules after a failure has occurred; periodic testing of modules (also known as "flexing" or "roving") to verify that they are still functional.

c. Software-Oriented Test and Recovery

A relatively unexplored technique which shows some promise of improving overall system reliability is the use of fault-tolerant software [31, 32]. With respect to software, fault tolerance implies three distinct functions: the ability to check the results of a computation (including the ability and intelligence to discriminate between faulty and fault-free computations), the ability to perform computation recovery, and the ability to reconfigure software. This technique necessarily requires a high degree of sophistication, and, despite potentially high payoffs, work in this field is still at the pioneering stage.

As in any implementation of fault tolerance, the first requirement for realizing software fault tolerance is the ability to recognize faulty computations. This ability demands a high degree of forethought and has the best probability of success when made a primary design goal. Some of the conceivable means available for checking the validity or correctness of a software module in real time include the use of watch-dog timers, address-in-bounds checks, and executable specification assertions.

A watch-dog timer may be included as an independent timing element which clocks and monitors the execution time of modules, interrupts, loops, or other program entities. Thus, it is possible to recognize suspicious processes by the amount of time consumed in the process. A process which is malfunctioning (i.e., violating its intended function) frequently performs futile computations or finds itself caught within an infinite loop. Wildly looping software or

"dead" processes (e.g., processes which wait on another hardware or software module that never responds) will then cause the watch-dog timer to timeout or attempt to interrogate the suspected process.

The address-in-bounds check detects the illegal use of addresses. Data and program addresses are often constrained to certain zones of the address space. A simple algorithm can then check that data and instruction addresses fall within the range which corresponds to these particular data and instructions. MCF architecture definition supports hardware checks for address-in-bounds by the memory management scheme [6].

A common means of testing module validity is the dynamic assertion. Certain programming languages (e.g., Ada, the primary MCF high-order language) allow the inclusion of predicates placed at entry and exit points of a module. The use of assertions in conjuntion with program correctness proofs provides a method for on-line checking to see that a program module meets its formal specification. Assertions offer a form of software redundancy; the failure of an assertion (i.e., when the asserted predicate tests false) during program execution is sufficient to indicate the existence of a fault in either the runtime module or the hardware associated with the module.

Once a fault has been recognized by the above or other means, some form of recovery is possible. For software exceptions, the analogue of the RETRY after a hardware exception is the idea of computation recovery (also known as backward recovery, rollback-and-retry). RETRY enables the program to continue execution after discovery of a fault. In this scheme, it is necessary to establish recovery points at various locations in the execution of the program. This is done by saving selected data or register values at the various points. For instance, a core image can be written from memory to backup storage, enabling the faulted program to back up and reattempt to execute the procedure(s) following the recovery point. Returning to a recovery point also opens the way for software reconfiguration, which will be discussed later. The costs incurred by computation recovery merit closer analysis since significant space could be required for saving the state of the computation at a recovery point and since the amount of time spent in retrying the faulted computation sequence could possibly dominate the cost of computation. In light of these factors, it would be worthwhile to explore cost-saving measures for computation recovery.

Instructions that support automatic recovery would be very effective. For instance, an instruction that designates a recovery point, RECOVERYPOINT #N, when executed would save some suitable image of the computation process in a recovery cache. Executing RECOVERYPOINT #N, AddrList would force the current contents of the recovery cache, recovery point N-1, into a main or secondary backing store, (see Figure 8). An instruction, RECOVER #N, Addr_ist, would reload the computation image saved at recovery point N. This would not change the recovery cache. In this way, if RECOVERY needs to be done again, the current image is still in cache; if RECOVERY on a prior point is required, the recovery cache is changed to reflect the image of that prior point and all intervening images are destroyed.

There are several issues that have not been discussed such as what exactly is a recovery cache, what happens if there is no recovery point N, and what is a "suitable image of the computation process" [33]. They will not be discussed here. It is important to realize that the instructions mentioned are only a few of the ones that might be required and many issues in "recovery" have been ignored here. With a suitable set of instructions several different recovery schemes, e.g., recovery blocks [29], can be more easily and efficiently implemented.

Software reconfiguration is similar in concept to hardware reconfiguration, but instead of replacing a faulty unit with a good copy of the unit, it attempts to replace a faulty program module with an alternate version of the module's function [32]. After fault detection and rollback, alternate versions of the re-executed modules may be invoked in place of the originals. From the recovery point to the point of program error there may be several different procedures or modules. Therefore, there exist several different sequences of originals or alternates that could be invoked. A simple strategy is to replace single modules by their alternates, testing each time at the point of failure, then replacing couples of modules if the fault is still present, etc. Since this involves a potentially large number of rollbacks, close attention should be paid to the costs sustained by such reconfiguration strategies. The cost of programming multiple versions of a function module is also a serious concern [29, 34].

Fig. 8. Recovery Cache Block Diagram.

4. CONCLUSION

The objective of this effort was to identify ways in which BIT can be integrated into the MIL-STD-1862 architecture very early in the development cycle. Since this is a radical departure from the classical approach to computer testing, significant original work had to be done to identify error-detecting approaches and ways to evaluate their corresponding effectiveness. This study led to reporting mechanisms, instruction retry, error recovery strategies, and finally to an overview of fault-tolerant software.

The initial problem was to characterize BIT-detected errors and to compare these characteristics to MIL-STD-1862 exceptions and MIL-STD-1862 interrupts. Some of the confusion in MIL-STD-1862 was cleared up in this respect. An approach for reporting BIT-detected errors to software was identified. The recommended reporting mechanism uses the current MIL-STD-1862 interrupt facility. This recommendation was based on the similarities between MIL-STD-1862 interrupts and BIT-detected errors. This is not to imply they are the same -- only that they share many characteristics in common.

The ability to invoke a software handler upon reciept of a BIT signal led to a discussion about the probable actions that might be required after correctly handling this type of error. The upshot of this was the recommendation that the handler be designed to return to the point in the computation process at which the error occurred. This is intimately tied in with the required capability to resume an instruction after handling an interrupt. Several instructions that explicitly control retry or resumption were discussed.

Looking at BIT-detected errors in a more general context, it is obvious that an overall strategy for recovery from BIT-detected errors is mandatory. This belief is based on the knowledge that the most common and dangerous fault is the transient fault, which appears and just as quickly disappears from the system. The proposed comprehensive recovery strategy involves a combination of both hardware and software working in concert.

Nonconcurrent BIT was also considered in this study. The idea of a system-wide diagnostic task was presented where each BIT handler was a subtask of a larger task. Another subtask of the diagnostic task was that of building,

maintaining and querying an "error data base." This error data base is updated everytime a BIT detected error is signaled. This data base can be queried by the BIT handlers or other subtasks for information concerning the previous history of specific modules and specific BIT errors. With this information an intelligent test and recovery strategy can be determined. This data base can also be used to correlate information for use by maintenance personnel. One strategy that might be proposed based on an analysis of the error data base would be to run diagnostics on the system or certain modules. Based on this it could be decided to test a module at a deeper level by executing module specific test patterns. These patterns would be read into a module and the module's response would be compared to some "gold standard." This is a hierarchy of tests that can be run from a diagnostic subtask, e.g., a BIT handler, to quickly determine if the module can be considered usable. Low-level hardware fault tolerance has been suggested using BIT and hardware retry to recover from most transient errors. For software exceptions, the analogue of the retry after a hardware exception is the idea of computation recovery. This enables the program to continue execution after a software exception has been raised. An example set of instructions are presented that could be used in a computation recovery scheme. The best work so far has been done by Lee [33], who explores the entire recovery mechanism in greater detail.

A great deal of follow-on work needs to be done on both the very practical problem areas of developing a comprehensive test plan for the machine implementations, as well as looking at extensions to the instruction set in the areas of BIT error handlers, rollback-and-recovery, and fault isolation. A comprehensive test plan needs to be evolved that addresses:

1. the effectiveness of vendor's BIT,
2. the adherence to architectural specification when BIT signals are raised,
3. raising the proper exceptions when the architectural specifications are violated, and
4. testing the exceptional conditions defined by the operating system.

The instruction set should be studied to determine if further instructions could be added that would be useful for:

1. handling BIT-detected errors,
2. testing modules for fault isolation from a system diagnostic task, and
3. implementing a comprehensive rollback-and-recovery scheme.

RTI, in cooperation with Carnegie-Mellon University personnel, has developed a BIT evaluation tool using a new ISP fault injection simulator. It is recommended that this tool now be applied to candidate MCF embodiments to aid the government in creating a maintainable MCF design to minimize future system life cycle maintenance costs.

REFERENCES

1. Prime Item Development Specification for MCF Super-Minicomputer AN/UYK-41, CORADCOM CR-CS-0034-001, June, 1980.

2. Prime Item Development Specification for MCF Microcomputer AN/UYK-49, CORADCOM CR-CS-0035-001, June, 1980.

3. Clary J.B., et al., "A Preliminary Study of Built-in-Test for the Military Computer Family," CORADCOM-76-0100-F, March, 1979.

4. Clary, J.B., et al., "Development of a Methodology for Verifying Military Computer Built-in-Test Performance Specifications," CORADCOM-80-0780-F, September, 1980

5. Barbacci, M.B., et al., "The ISPS Computer Description Language," Department of Computer Science, Carnegie-Mellon University, August, 1979.

6. "Instruction Set Architecture for the Military Computer Family," MIL-STD-1862, 20 May, 1980.

7. Clary, J.B. and R.A. Sacane, "Self-Testing Computers," Computer, October, 1979.

8. Ball, M. and F. Hardie, "Effects and Detection of Intermittent Failures in Digital Systems," Spring Joint Computer Conference, 1969.

9. Rennels, D.A., "Distributed Fault-Tolerant Computer Systems," Computer, March 1980.

10. Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," Communications of the ACM, December, 1975.

11. Mitchell, J.G., W. Maybury, and R. Sweet, "Mesa Language Manual," Xerox PARC CSL-79-3, April, 1979.

12. Levin, R., "Program Structures for Exceptional Condition Handling," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, June, 1977.

13. BLISS Language Guide, Digital Equipment Corporation, September, 1978.

14. Ichbiah, J.D., J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, and B.A. Winchmann, "Preliminary ADA Reference Manual and Rationale," ACM SIGPLAN Notices, v. 14, no. 6, parts A and B, June, 1979.

15. STEELMAN Requirements for High Order Computer Programming Languages, DoD, June, 1978.

16. Fuller, S.H., H.S. Stone, and W.E. Burr, "Initial Selection and Screening of the CFA candidate architectures," AFIPS, 1979.

REFERENCES
(Continued)

17. Prime Item Development Specification for MCF Computer Control Panel, CORADCOM CR-CS-0036-001, June, 1980.

18. Personal communications with Dietz and Szewerenko.

19. Sellers, F., M. Hsiao, and L. Bearnson, Error Detecting Logic for Digital Computers, McGraw-Hill Book Company, 1968.

20. Sedmak, R.M. and H.L. Liebergot, "Fault-Tolerance of a General Purpose Computer Implemented by Very Large Scale Integration," Proceedings of FTCS-8, June, 1978.

21. Carter, W. et. al., "Cost Effectiveness of Self Checking Computer Design," Proceedings of FTCS-7, June, 1977.

22. Liskov, B., and A. Snyder, "Exception Handling in CLU," IEEE Transactions on Software Engineering, v. SE-5, no. 6, November, 1979.

23. Wuerges, H., "Reaktion auf Unerwuenschte Ereignisse in Hierarchisch Strukturten Software-Systemen," Dr. Phil. Thesis, Technische Hochschule Darmstadt, (Translation by D. Parnas) FRG, November, 1977.

24. McConnel, S. and D. Siewiorek, "C.vmp: The Implementation, Performance, and Reliability of a Fault Tolerant Multiprocessor," Departments of Electrical Engineering and Computer Science, CMU-CS-78-108, Carnegie-Mellon University, March, 1978.

25. Ng, Y. and A. Avizienis, "Reliability Modeling and Prediction for Fault-Tolerant Digital Systems," Technical Report (draft), Department of Computer Science, UCLA, January, 1979.

26. Carter, W., et. al., "Logic Design for Dynamic and Interactive Recovery," IEEE-TC, November, 1971.

27. Morgan, D.E. and D.J. Taylor, "A Survey of Methods of Achieving Reliable Software," Computer, February, 1977.

28. Meyers, G.J., Software Reliability Principles and Practices, Wiley-Interscience Publication, 1976.

29. Randell, B., P.A. Lee, and P.C. Treleaven, "Reliability Issues in Computing System Design," ACM Computing Surveys, June, 1978.

30. VAX11/780 Architecture Handbook, Digital Equipment Corporation, 1977.

31. Hecht, H., "Fault-Tolerant Software for Real-Time Applications," ACM Computing Surveys, v. 8, no. 4, December, 1976.

REFERENCES
(Continued)

32. Kim, K.H., "Error Detection, Reconfiguration and Recovery in Distributed Processing Systems," Proceedings of the Distributed Computing Systems, October, 1979.

33. Lee, P. A., N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," FTCS-9, June, 1979.

34. Chen, A. and A. Avizienis, "N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation," FTCS-8, June, 1978.

APPENDIX A

MIL-STD-1862.ISP Description

```
Nebula :=
BEGIN
**Machine.State**
!Processor Status Word
Kernel<> :=          PSW<0>,                    !Kernel/ User mode
Last.mode<> :=       PSW<1>,                    !Previous context (Kernel/Task)
Pri<0:4> :=          PSW<4:8>,                  !Processor priority
C<> :=               PSW<9>,                    !Carry condition code
T<> :=               PSW<10>,                   !Truncate cc
N<> :=               PSW<11>,                   !Negative (less) cc
Z<> :=               PSW<12>,                   !Zero cc
Debug<0:1> :=        PSW<13:14>,                !Debugging Control
Privilege<> :=       PSW<15>,                   !Privileged if set
Base<> :=  PSW<16>,                        !Base of context stack
Superv<> :=          PSW<17>,                   !Supervisor/Task mode
UDLE<> :=  PSW<18>,                        !Up/Down level exception
EAE<> :=   PSW<19>,                        !Exception on Arithmetic Error
MaxReg<0:3> :=       PSW<20:23>,                !Number of registers in current
 context
MaxPar<0:7> :=       PSW<24:31>,                !Number of parameters in current
 context
PC<0:31>,                                  !Program Counter
!Machine State Registers
Ctxp[0:1]<0:31> := MSR[0:1]<0:31>,          !Context Pointers
 Kctxp<0:31> :=              Ctxp[0]<0:31>,          !Kernel
 Tctxp<0:31> :=              Ctxp[1]<0:31>,          !Task
Soft.int.req<0:31> :=        MSR[2]<0:31>,          !Software Interrupt
 Request
PSW<0:31> :=                 MSR[3]<0:31>,          !Processor Status
Vreg[0:3]<0:31> := MSR[4:7]<0:31>,          !SVC and OPEX vector registers
ASR<0:31> :=                 MSR[8]<0:31>,          !Auxillary Status
 Register
  IC<> :=            ASR<18>,                   !Infinity Control
  MI<> :=            ASR<19>,                   !Mask for Invalid Operand
  MQ<> :=            ASR<20>,                   !Mask for Division by Zero
  MO<> :=            ASR<21>,                   !Mask for Overflow
  MU<> :=            ASR<22>,                   !Mask for Underflow
  MP<> :=            ASR<23>,                   !Mask for Inexact
  RC<0:1> :=                 ASR<25:26>,             !Rounding Control
  I<>  :=            ASR<27>,                   !Floating Pt. Invalid operand
  Q<>  :=            ASR<28>,                   !Floating Pt. Division by Zero
  O<>  :=            ASR<29>,                   !Floating Pt. Overflow
  U<>  :=            ASR<30>,                   !Floating Pt. Underflow
  P<>  :=            ASR<31>,                   !Floating Pt. Inexact
MMreg[0:1]<0:31> :=          MSR[9:10]<0:31>,       !Memory management
```

```
 registers
Timctl<0:31> :=              MSR[11]<0:31>,          !Timer Control Reg
Inttim[0:3]<0:31> :=         MSR[12:15]<0:31>,       !Interval timers
TOD<0:31> :=                 MSR[16]<0:31>,          !Time of Day
!
!Exception codes
!
Macro Spec.error :=          |1| ,
Macro Ill.Mode :=            |2| ,
Macro Ill.Param :=           |3| ,
Macro Ill.Reg :=             |4| ,
Macro Ill.Write :=           |5| ,
Macro Ill.Size  :=           |6| ,
Macro Ill.Addr  :=           |7| ,
Macro Bad.displacement :=    |8| ,
Macro Context.Alignment :=   |9| ,
Macro Context.Base :=        |10| ,
!Arithmetic Group
Macro Ill.Divisor :=         |16| ,
Macro Truncate :=            |17| ,
Macro Range.error :=         |18| ,
Macro Ill.Operation :=       |19| ,
Macro Div.by.Zero :=         |20| ,
Macro Overflow :=            |21| ,
Macro Underflow :=           |22| ,
Macro Inexact :=             |23| ,
Macro Unordered :=           |24| ,
Macro Task.Failure :=        |32| ,
Macro Break :=               |33| ,
Macro Inst.Trace :=          |34| ,
Macro Proc.Trace :=          |35| ,
!
!Vectors
!
!Physical
Macro      SI.vec :=         |"4| ,   !Software interrupt vector
Macro      PI.vec :=         |"8| ,   !Parameterized Interrupts
Macro      MM.vec :=         |"C| ,   !Memory Management Errors
Macro      ME.vec :=         |"10| ,  !Memory system errors
                  !Hard error "14, Soft error "10
Macro      PF.vec :=         |"18| ,  !Power failure
Macro      PR.vec :=         |"1C| ,  !Power restore
Macro      Kernel.Save :=    |"20| ,  !Pointer to kernel save area
Macro      Exception.vec:=   |"24| ,  !Supervisor exception handler
Macro      Priv.error :=     |"28| ,  !Privileged instruction in user mode
!Macro Timer0.Vec :=         |"30| ,  !Timer 0 Vector
!Macro Timer1.vec :=         |"34| ,  !Timer 1 Vector
!Macro Timer2.Vec :=         |"38| ,  !Timer 2 Vector
!Macro Timer3.Vec :=         |"3C| ,  !Timer 3 Vector
!?!The following macros define the values of operand type and
!?!size used in the ISP implementation.
!?!Operand Sizes (op.addr<Op.Size>)
!Note that these are visible as the size fields of operands.
Macro      Dwrd :=  |'00| ,                !8 bytes
```

```
Macro       Byte := | '01 ,                    !1 byte
Macro       Hwrd := | '10 ,                    !2 bytes
Macro       Word := | '11 ,                    !4 bytes
!?!Operand Types (op.addr<Op.Type>)
Macro       Constant := | '00 ,                !?!Descriptor IS the operand
Macro       Context := | '01 ,                 !?!Register or in context stack
Macro       Literal := | '10 ,                 !?!Literal in code stream
Macro       Memory  := | '11 ,                 !?!In Memory
Macro       single := | 0 ,                    !single size
Macro       double := | 1 ,                    !Double size
!Size converts an operand size into the equivalent number of bytes
Size(ops<1:0>)<4:0> :=
   BEGIN
   DECODE ops =>
           BEGIN
           Dwrd := size = 8,
           Byte := size = 1,
           Hwrd := size = 2,
           Word := size = 4
           END
   END,
!Sign extend a value VAL of size OPS to 64 bits
sxt(val<63:0>,ops<1:0>)<63:0> :=
   BEGIN
   DECODE ops =>
           BEGIN
           Dwrd := sxt = val,
           Byte := sxt <= val<7:0>,
           Hwrd := sxt <= val<15:0>,
           Word := sxt <= val<31:0>
           END
   END,
Macro       Op.type := | 35:34 ,
Macro       Op.size := | 33:32 ,
**Memory.Access** (US)
Macro MaxMem := | 4095 ,                       ! Model Dependent
!Main Memory
Mb[0:MaxMem]<0:7>,
Mw[0:MaxMem]<0:31>(increment: 4) :=          Mb[0:MaxMem]<0:7>,
Md[0:MaxMem]<0:63>(increment: 8) :=          Mb[0:MaxMem]<0:7>,
!I/O Space
!MBio["FFF00000:"FFFFFFFF]<0:7>,    !Full I/O space is 20 bits physical
MACRO IO.Min :=     | "FFFFF000 ,     !Small amount defined for ISP's purpose
!I/O space may be accessed on ALIGNED ADDRESSES as byte, half, word, double
MBio[IO.Min:"FFFFFFFF]<0:7>,
MHio[IO.Min:"FFFFFFFF]<0:15>(increment: 2) := MBio[IO.Min:"FFFFFFFF]<0:7>,
MWio[IO.Min:"FFFFFFFF]<0:31>(increment:4):= MBio[IO.Min:"FFFFFFFF]<0:7>,
MDio[IO.Min:"FFFFFFFF]<0:63>(increment:8):= MBio[IO.Min:"FFFFFFFF]<0:7>,
MSR[0:16]<0:31> := MBio["FFFFF800:"FFFFF843]<0:7>,
!Memory read routine
read(va<31:0>,ops<1:0>,a<2:0>)<63:0> :=
    BEGIN
    DECODE vp(va,a)<0:11> EQL "FFF =>
   BEGIN
```

```
    !Memory
    DECODE ops =>
        BEGIN
    Dwrd := read = Mb[vp]@Mb[vp(va+1,a)]@Mb[vp(va+2,a)]@Mb[vp(va +3,a)]@
             Mb[vp(va+4,a)]@Mb[vp(va+5,a)]@Mb[vp(va+6,a)]@Mb[vp(va+7,a)],
    Byte := read = Mb[vp],
    Hwrd := read = Mb[vp]@Mb[vp(va+1,a)],
    Word := read = Mb[vp]@Mb[vp(va+1,a)]@Mb[vp(va+2,a)]@Mb[vp(va+3,a)]
        END,
    !I/O Space
    DECODE ops =>
        BEGIN
    Dwrd := read = MDio[vp],
    Byte := read = MBio[vp],
    Hwrd := read = MHio[vp],
    Word := read = MWio[vp]
        END
    END
     END,
!Memory write routine
write(va<31:0>,ops<1:0>,a<2:0>)<63:0> :=
     BEGIN
     DECODE vp(va,a)<0:11> EQL "FFF =>
    BEGIN
    !Memory
    DECODE ops =>
        BEGIN
    Dwrd := Mb[vp]@Mb[vp(va+1,a)]@Mb[vp(va+2,a)]@Mb[vp(va+3,a)]
            Mb[vp(va+4,a)]@Mb[vp(va+5,a)]@Mb[vp(va+6,a)]@Mb[vp(va+7,a)]=write,
    Byte := Mb[vp] = write,
    Hwrd := Mb[vp]@Mb[vp(va+1,a)] = write,
    Word := Mb[vp]@Mb[vp(va+1,a)]@Mb[vp(va+2,a)]@Mb[vp(va+3,a)] = write
        END,
    !I/O Space
    DECODE ops =>
        BEGIN
    Dwrd := MDio[vp] = write,
    Byte := MBio[vp] = write,
    Hwrd := MHio[vp] = write,
    Word := MWio[vp] = write
        END
    END
    END,
**Address.Translation**(us)
Mptr[0:1]<0:31> := MMreg[0:1]<0:31>,       !Map address registers
!Fields in Mptr registers
MACRO Map.addr := |1:28|,          !address of map (bits 29:31 are 0)
!Bit 29 is reserved
```

```
MACRO Map.reloc := | 30 ,          !set if relocation enabled
MACRO Map.prot := | 31 ,           !set if protection enabled
M.ent<0:63>,       !Map entry temporary
!Map entry fields
M.priv<> :=                M.ent<0>,       !Privilege
M.bound<0:27> :=   M.ent<1:28>,    !Virtual Address Bound
M.prot<0:2> :=             M.ent<29:31>,   !Protection Key
M.reloc<0:28> :=   M.ent<32:60>,   !Relocation Amount
Macro M.Maxp :=    | 4 ,    !Implementation dependent seg. number size
Seg.Max<0:M.Maxp>, !Maximum Seg number temporary
M.addr<0:31>,              !Map address temporary
!Memory Access Codes
MACRO M.c :=       | 0 ,    !Context Access
MACRO M.i :=       | 1 ,    !Instruction fetch
MACRO M.r :=       | 2 ,    !Memory read
MACRO M.w :=       | 3 ,    !Memory read/write
MACRO M.n :=       | 4 ,    !No Access
!Fault error codes
MACRO Inv.sup :=   | 1 ,    !Invalid access to supervisor space
MACRO Inv.Seg :=   | 2 ,    !No Seg containing this address
MACRO Inv.accs :=  | 3 ,    !Access type violation
MACRO Priv.violation :=    | 4 ,
!
!Virtual to physical Address translation
!
vp(va<0:31>,mode<0:2>)<0:31> :=
   BEGIN
   vp = va NEXT
   IF va<0> AND NOT Superv=> M.fault(Inv.Sup) NEXT !User access to superv
   IF Mptr[Va<0>]<30:31> => Seg.number(va) NEXT
   IF Mptr[Va<0>]<Map.Prot> =>     !Protection Enabled
       BEGIN
           IF M.Ent<M.priv> AND NOT Privilege=> M.fault(Priv.Violation)NEXT
           DECODE mode =>
               BEGIN
           M.c:=   IF M.prot NEQ '011 => M.fault(Inv.accs),
           M.i:=   IF M.prot<0:1> NEQ '10 => M.fault(Inv.accs),
           M.r:=   IF M.prot<1:2> NEQ '01 AND
                           M.prot NEQ '010 => M.fault(Inv.accs),
           M.w:=   IF M.prot NEQ '010 => M.fault(Inv.accs),
           M.n:=   No.Op()
               END
       END NEXT
   DECODE Mptr[Va<0>]<Map.Reloc> =>
       BEGIN
           vp<0:4> <=(tc) va<0> AND va<5:11> EQL "7F, !Fix for I/O Space
           vp = va<5:31> + M.reloc@'000            !Relocation Enabled
       END
   END,
Seg.number(va<0:31>)<0:M.Maxp> :=
   BEGIN
   Seg.number = 0;
   M.addr = Mptr[va<0>]<Map.addr>@'000 NEXT
   Seg.max = Mw[M.addr-4] NEXT
```

```
   REPEAT BEGIN
       M.ent = Md[M.addr] NEXT      !Get map entry
       IF va<1:29> LEQ M.bound => LEAVE Seg.number NEXT
       Seg.number = Seg.number + 1;
       M.addr = M.addr + 8 NEXT
       IF Seg.number GTR Seg.max => M.fault(Inv.Seg)
       END
   END,
M.fault(fcode<0:15>)<0:15> :=
   BEGIN
   M.fault = fcode; MMf = 1 NEXT    !Set fault
   RESTART Run                      !And abort
   END,
**Context.Stack.Access**  (US)
! Most of this section is implementation dependent
!The context stack contains the context of the currently running process.
!The top of the stack contains the current process registers.
!This stack is restricted to word (4 byte) boundaries.
!?!In most implementations the top of stack will be cached in some way.
!?!The ISP reflects one very simple mechanism.
!Context stack read
Read.Ctx(disp<31:0>)<31:0> :=
   BEGIN
   DECODE disp =>
           BEGIN
           0 :=    Read.Ctx = PC,          ! PC is cached in the ISP
           Otherwise :=    Read.Ctx = Mw[vp(Ctxp[Kernel] + disp@'00, M.c)]
           END
   END,
! Context Stack Write
Write.Ctx(Disp<31:0>)<31:0> :=
   BEGIN
   Mw[vp(Ctxp[Kernel] + Disp@'00, M.c)] = Write.Ctx
   END,
Reload.Ctx<31:0> :=
   BEGIN
   PC = Mw[vp(Ctxp[kernel], M.c)] NEXT
   Reload.ctx = Mw[vp(Ctxp[Kernel]-4, M.c)]
   END,
**Call.mechanism**(us)
!
!Call.switch implements the procedure call mechanism with parameter passing
!
P.Var<> :=          Call.Switch<15>,
P.Exception<> :=    Call.Switch<14>,
MaxReg.New<3:0> := Call.Switch<11:8>,
MaxPar.New<7:0> := Call.Switch<7:0>,
pcount<31:0>,                               !Parameter counter temp
Ctx.size<31:0>,                             !New context size temp
Call.switch(addr<31:0>)<15:0> :=
   BEGIN
   call.switch = read(addr,Hwrd,M.i) NEXT  !get procedure descriptor
   addr = addr + 2;
```

```
    !Determine number of parameters
    IF P.Var =>                     ! variable number of parameters?
            BEGIN
            MaxPar.New = get.log() NEXT
            IF get.log GTR 255 => Exception(Ill.Param)
            END NEXT
    !Determine size of new context
    Ctx.size = Maxpar.new + Maxreg.New + 3 NEXT
    write.ctx("FFFFFFFF) = PSW NEXT         ! Save current PSW
    !Set exception handler for this context
    DECODE P.Exception =>
            BEGIN
            0 :=    write.ctx = 0,
            1 :=    BEGIN
                    write.ctx = read(addr,Hwrd,M.i) + addr NEXT
                    addr = addr + 2
                    END
            END NEXT
    write.Ctx("FFFFFFFE) NEXT               ! Save Exception Handler
    !Evaluate the Parameters
    pcount = 0 NEXT
    loop := REPEAT
            BEGIN
            IF pcount EQL MaxPar.New => LEAVE loop NEXT
            IF op.addr()<op.type> EQL Context =>    !Ref to prior register
                    op.addr<31:0> = op.addr<31:0> + Ctx.size NEXT
            Write.ctx(pcount-Maxpar.new-2)=op.addr<35:31>@op.addr<26:0> NEXT
            pcount = pcount + 1
            END NEXT
    !complete the control transfer
    write.ctx(0) = PC NEXT          ! Save current PC
    !check existance to -Ctx.size -1 (new PSW storage)
    Ctxp[Kernel] = Ctxp[Kernel] - Ctx.size@'00 NEXT  ! Point to new context
    IF MaxReg GTR 0 AND MaxReg.New GTR 0 =>         ! Copy "SP"
            write.ctx(1) = read.ctx(Ctx.size+1) NEXT        !if both exist
    PC = addr;                      ! New PC
    PSW<18:31> = Call.Switch;       ! New PSW
    Base = 0; Superv = Superv AND addr<31>
    END,
!
!Build.stack initiates a call stack on the current stack
!
!descriptor is same format as for Call.stack
B.exception<> :=   Build.stack<14>,
B.maxreg<3:0> :=   Build.stack<11:8>,
Build.stack(inc<31:0>,addr<31:0>,np<7:0>)<15:0> :=
    BEGIN
    Build.stack = read(addr,Hwrd,M.i) NEXT
    addr = addr + 2;
    Build.stack<7:0> = np;  !Number of parameters
    Ctxp[Kernel] = Ctxp[Kernel] - (B.maxreg + np + 3 - inc)@'00 NEXT
    !install exception handler
```

```
    DECODE B.exception =>
        BEGIN
            write.ctx = 0,  !none
            BEGIN
                write.ctx = read(addr,hwrd,M.i) + addr NEXT
                addr = addr + 2
            END
        END NEXT
    Write.ctx(B.maxreg+np+1) NEXT
    !Set psw
    PSW<18:31> = Build.stack; Superv = addr<31>;Debug = 0;
    PC = addr
    END,
!
!Call.restore removes a context frame from the context stack
!
Call.restore :=
    BEGIN
    Ctxp[Kernel] = Ctxp[Kernel] + (Maxreg + MaxPar + 3)@'00 NEXT
    DECODE Base =>
        BEGIN
            psw<13:31> = Reload.ctx(),
            BEGIN
                Kernel = last.mode NEXT
                psw = Reload.ctx()
            END
        END
    END,
!
!Pop.stack removes an entire execution stack from the context stack
!
Pop.stack :=
    BEGIN
    REPEAT BEGIN
        Ctxp[Kernel] = Ctxp[Kernel] + (Maxreg+MaxPar+3)@'00 NEXT
        IF Base => LEAVE Pop.stack NEXT
        PSW = Read.ctx("FFFFFFFF)
        END
    END,
**Operand.Descriptors** (US)
!Operand descriptor routine
!Reads an operand specifier from instruction stream and generates
!A descriptor specifying its type, size, and address
op.addr<35:0> :=
BEGIN
**Local.Declarations**
op.spec<7:0>,                                   !operand specifier
Parameter(Num<7:0>)<35:0> :=
    BEGIN
    IF Num GTR(us) MaxPar => Exception(Ill.param) NEXT
    DECODE Num NEQ 0 =>
            BEGIN
            Of :=   parameter = MaxPar,
            1t :=   BEGIN
```

```
                    Read.Ctx(Num + Maxreg) NEXT
                    Parameter = Read.Ctx<31:27>@"0@Read.Ctx<26:0>
                    END
            END
    END,
op.decode<7:0> :=
    BEGIN
    op.spec = read(pc,byte,M.i) NEXT
    pc = pc + 1 NEXT
    DECODE op.spec =>
            BEGIN
                                    !Short Literal Mode
    '000????? :=    op.addr = op.spec,      !Constant
    '0010???? :=    BEGIN           !Register Mode
                    IF op.spec<3:0> EQL 0 OR op.spec<3:0> GTR Maxreg =>
                                    Exception(Ill.Reg) NEXT
                    op.addr = Context@Word@"0000000@op.spec<3:0>
                    END,
                                    !Short Parameter Mode
    '00110??? :=    op.addr = parameter(op.spec<2:0>),
    '001110?? :=    BEGIN           !Recursive Modes (parameter, indexed)
                    IF op.decode NEQ 0 => Exception(Ill.Mode) NEXT
                    op.decode = op.spec     !Save for recursive evaluation
                    END,
    '001111?? :=    BEGIN           !Literal Mode
                    op.addr = Literal@op.spec<1:0>@pc NEXT
                    pc = pc + size(op.spec<1:0>)
                    END,
    '??000000 :=    BEGIN           !Absolute Address
                    op.addr= Memory@op.spec<7:6>@read(pc,Word,M.i)<31:0>NEXT
                    pc = pc + 4
                    END,
    '??00???? :=    BEGIN           !Register Indirect
                    IF op.spec<3:0> GTR Maxreg => Exception(Ill.Reg) NEXT
                    op.addr = Memory@op.spec<7:6>@Read.Ctx(op.spec<3:0 >)
                    END,
    OTHERWISE :=    BEGIN           !Indexed Memory Modes
                    IF op.spec<3:0> GTR Maxreg => Exception(Ill.Reg) NEXT
                    read(pc,op.spec<5:4>,M.i) NEXT  !Index
                    sxt(read,op.spec<5:4>) NEXT     !SignExtend
                    op.addr<35:32> = Memory@op.spec<7:6>;
                    op.addr<31:0> = sxt + Read.ctx(op.spec<3:0>) NEXT
                    pc = pc + size(op.spec<5:4>)
                    END
            END
    END,
MAIN entry :=
    BEGIN
    op.decode = 0 NEXT
    IF op.decode() EQL 0 => LEAVE op.addr NEXT      !Non-recursive modes
    DECODE op.decode() =>
            BEGIN
```

```
    '00111000 :=    BEGIN   !Parameter
                    IF fetch(op.addr,0)<63:8> => Exception(Ill.param) NEXT
                    op.addr = parameter(fetch)
                    END,
    '00111001 :=    BEGIN   !Unscaled Index
                    fetch(op.addr,1) NEXT   !Index
                    op.decode() NEXT        !Base
                    op.addr<31:0> = op.addr + fetch
                    END,
    '00111010 :=    BEGIN   !Scaled Index Single Length
                    fetch(op.addr,1) NEXT   !Index
                    op.decode() NEXT        !Base
                    op.addr<31:0>=op.addr + fetch*size(op.addr<Op.Size>)
                    END,
    '00111011 :=    BEGIN   !Scaled Index Double Length
                    fetch(op.addr,1) NEXT   !Index
                    op.decode() NEXT        !Base
                    op.addr<31:0>=op.addr + fetch*size(op.addr<Op.Size>)*2
                    END
            END
    END
END,
**Access.by.Descriptor**
!fetch operand
!op.adr - descriptor of operand
!s - set for sign extend
fetch(op.adr<35:0>,s<>)<63:0> :=
    BEGIN
    DECODE op.adr<Op.type> =>
            BEGIN
            Constant:= Fetch = op.adr<31:0>,
            Context := Fetch = Read.Ctx(op.adr),
            Literal := Fetch = read(op.adr,op.adr<Op.size>,M.i),
            Memory := Fetch = read(op.adr,op.adr<Op.size>,M.r)
            END NEXT
    IF s => fetch = sxt(fetch,op.adr<op.size>)
    END,
Store(op.adr<35:0>)<63:0> :=
    BEGIN
    DECODE op.adr<op.type> =>
            BEGIN
            Constant := Exception(Ill.write),
            Context := Write.Ctx(op.adr) = store,
            Literal := Exception(Ill.write),
            Memory  := Write(op.adr,op.adr<op.size>,M.w) = store
            END
    END,
**Operand.Access**
Get.int(d<>)<63:0> :=
    BEGIN
    op.addr() NEXT
    IF d AND op.addr<op.type> EQL Memory =>
            op.addr<op.size> = op.addr<op.size> + 1 NEXT    !double size
    get.int = fetch(op.addr,1)
```

```
    END,
Put.int(d<>,repl<>)<63:0> :=
    BEGIN
    IF NOT repl =>
            BEGIN
            op.addr() NEXT
            IF d AND op.addr<op.type> EQL Memory =>
                    op.addr<op.size> = op.addr<op.size> + 1
            END NEXT
    Store(op.addr) = put.int;
    t = (put.int NEQ sxt(put.int,op.addr<op.size>)) NEXT
    n = sxt<63>;
    z = sxt EQL 0
    END,
Get.log<31:0> :=
    BEGIN
    op.addr() NEXT
    get.log = fetch(op.addr,0)
    END,
Put.log(repl<>)<31:0> :=
    BEGIN
    IF NOT repl => op.addr() NEXT
    store(op.addr) = put.log;
    n = sxt(put.log,op.addr<op.size>)<31> NEXT
    z = sxt EQL 0
    END,
Get.float<79:0> :=
     BEGIN
     op.addr() NEXT
     IF op.addr<35:32> EQL literal @ byte =>
            BEGIN
            get.float = special.case() NEXT
            LEAVE get.float
            END NEXT
    IF op.addr<op.type> EQL Memory =>
            op.addr<op.size> = op.addr<op.size> + 1 NEXT
    Get.float = unpack(fetch(op.addr,0),op.addr<op.size>)
    END,

special.case<79:0> :=
    Begin
    !Decode fetch(Op.Addr,0) =>
    !       Begin
    !       special.case = ?,
    !       special.case = ?
            !.....
    !       End
    no.op()
    End,
Put.float(repl<>)<79:0> :=
    BEGIN
    IF NOT repl =>
            BEGIN
            op.addr() NEXT
```

```
            IF op.addr<op.type> EQL Memory =>
                        op.addr<op.size> = op.addr<op.size> + 1
            END NEXT
      store(op.addr) = pack(put.float,op.addr<op.size>)
      END,
   Get.field(pos<31:0>,size<31:0>)<31:0> :=
      BEGIN(us)
      IF size GTR 32 => exception(ill.size) NEXT
      IF size EQL 0 => (get.field = 0; LEAVE get.field) NEXT
      op.addr<31:0> = op.addr()<31:0> +(tc) pos<31:3> NEXT    !Byte Address
      Memory.Chk( ) NEXT
      pos = op.addr<1:0>@pos<2:0> NEXT !Position from word boundary
      size = size + pos -1 NEXT                !End bit position
      read = MASK.LEFT(read(op.addr AND "FFFFFFFA,Word+size<5>,M.r), pos) NEXT
      get.field = read SRO (31-size<4:0>)
      END,
   Put.field(pos<31:0>, size<31:0>, repl<>)<31:0> :=
      BEGIN(us)
      IF size GTR 32 => exception(ill.size) NEXT
      IF size EQL 0 => LEAVE put.field NEXT
      put.field = MASK.LEFT(put.field, 32-size);
      IF NOT repl =>
          op.addr<31:0> = op.addr()<31:0> +(tc) pos<31:3> NEXT
      Memory.Chk() NEXT
      pos = op.addr<1:0>@pos<2:0> + size -1 NEXT       !position of end bit
      op.addr<1:0> = 0 NEXT
      read = read(op.addr, Word+pos<5>, M.r) SRR (31-pos) NEXT
      read = MASK.RIGHT(read, size) OR put.field NEXT
      write(op.addr, Word+pos<5>, M.w) = read SLR (31-pos)
      END,
   ** Instruction.Interpretation ** (US)
   ir<7:0>,                              !?!instruction register
   replace<>  := ir<0>,                  ! Result in last operand fetched
   long.branch<>      := ir<0>,                  ! 16 bit branch displacement
   pc.back<31:0>,                                ! initial PC for fault recovery
   Start :=
      BEGIN
      Soft.Int.Req = ASR = Exception = 0;
      !Ctxp = Vreg = MMreg = undefined<31:0>
      Power.up() NEXT
      Run()
      END,
   Run :=     BEGIN
      Int.Service() NEXT
      pc.back = pc;
      ir = read(pc,byte,M.i) NEXT
      pc = pc + 1 NEXT
      IEX() NEXT
      RESTART run
      END,
      REQUIRE.ISP | IEX.ISP| ,
   !
   !Exception handler
   !
```

```
Exception(ecode<29:0>)<> :=
BEGIN
    DECODE UDLE AND NOT Exception =>
   BEGIN
   0:=    BEGIN    !Upward to calling routines
           PC = read.ctx(maxreg+MaxPar+1) NEXT     !Get specified handler
           IF PC NEQ 0 =>          !If handler exists
               BEGIN
                   Write.ctx(maxreg+MaxPar+1) = ecode;     !save code
                   LEAVE Iex
               END NEXT
           IF NOT Base =>          !No handler but caller exists
               BEGIN
                   Call.restore();
                   RESTART Exception       !Try the caller
               END NEXT
           !No handler and bottom of stack
           Sup.eh(maxreg+MaxPar+3,Task.Failure) NEXT
           Base = 1        !This is all that's left
       END,
   1:=    Sup.eh(0,ecode)  !Down to the supervisor
   END NEXT
    LEAVE Iex
END,
Sup.eh(soff<0:31>,ecode<29:0>) :=  !Entry to supervisor exception handler
   BEGIN
   Write.ctx("FFFFFFFF) = PSW NEXT
   Write.ctx(0) = PC NEXT
   Superv = 1 NEXT
   Build.Stack(soff,Mw[Exception.vec]<0:29>@'00,3) NEXT
   Privilege = Mw[exception.vec]<31>;
   Write.ctx(Maxreg+1) = ecode NEXT
   Write.ctx(Maxreg+2) = Memory@Byte@pc.back<0>@pc.back<5:31> NEXT
   Write.ctx(Maxreg+3) = Context@(Maxreg+maxpar+3)<29:0>
   END,
OPEX :=            !Unimplemented opcode handler
BEGIN
    vector.call(0,ir)
END,
Vector.call(b<>,index<15:0>)<31:0> :=
BEGIN
    DECODE index LSS Vreg[b@'0]<0:15> =>
   BEGIN
   Of :=   index = index - Vreg[b@'0]<0:15>,
   lt :=   index = 0
   END NEXT
    IF index GTR Vreg[b@'0]<16:31> => index = 0;
    superv = privilege = 1 NEXT    !Full privilege for vector access
    Vector.call = Mw[vp( Vreg[b@'1] + index@'00, M.r)] NEXT
    Call.switch(vector.call<31:2>@'00) NEXT
    Privilege = Vector.call<0>; Debug = 0
END,
Fp.exception(ecode<4:0>) :=
    BEGIN
```

```
    DECODE ASR<ecode> =>
   BEGIN
   0  :=
       DECODE EAE =>
           BEGIN
           t = 1,
           exception(ecode)
           END,
   1  := ASR<ecode +(US) 8> = 1
   END
    END,
**Interrupt.Service**(us)
!?!A device may request an interrupt by storing its vector location
!?!in Ext.vec and setting the appropriate bit of Ext.int.vec. Note
!?!that this IMPLEMENTATION is for the convenience of the ISP and should
!?!not be taken literally
Ext.int.req<0:31>,          !?!External interrupt request
Ext.vec[0:31]<0:31>,                !?!External interrupt vector
!?!The following are set by the memory system when errors occur.
!?!A soft error will set these only if enabled in the ASR
!?!control register is set
MER<0:31>,                  !?!Address of failed memory location
HME<>,                              !?!Set if hard memory error
MMf<> :=    Int.service<0>, !?!Memory Management Fault
Mem.err<> :=        Int.service<1>, !?!Memory system hard or soft error
Pwr.fail<> :=       Int.service<2>, !?!Power failure
Rp.tmp<0:5>,        !Temp for priority
Int.Service<0:2> :=
   BEGIN
   !Internal interrupts
   IF Int.Service =>
       DECODE first.one(Int.Service) =>
           BEGIN
           0:= BEGIN       !Memory management fault
                   MMf = 0; trap(MM.vec,4) NEXT
                   !Fill in parameters
                   Write.ctx(Maxreg+1) = Memory@Byte@vp<31>@vp<26:  0>NEXT
                   Write.ctx(Maxreg+2) = Memory@Byte@PC.back<31>@
                                                       pc.back<26:0> NEXT
                   Write.ctx(Maxreg+3) = seg.number NEXT   !Segment number
                   Write.ctx(Maxreg+4) = M.fault           !Fault code
               END,
           1:= BEGIN       !Memory error
                   Mem.err = 0; trap(ME.vec+HME@'00,1) NEXT
                   Write.ctx(Maxreg+1) = MER<27:0>
               END,
           2:= BEGIN       !Power fail
                   pwr.fail = 0;
                   !?!Implementation shall flush all caches at this point
                   Mer = Mw[Kernel.save] NEXT
                   Mw[Mer] = Kctxp<0:30>@Kernel;    !Save kernel context
                   Mw[Mer+4] = Mptr[1];    !and supervisor map pointer
                   Trap(PF.vec,0) NEXT
```

```
                    Pri = "1F       !Priority to maximum
                END
            END NEXT
        !External Interrupts
        Rp.tmp = last.one(mask.left(Ext.int.req OR Soft.int.req,Pri+1)) NEXT
        IF Rp.tmp EQL 32 => LEAVE Int.service NEXT
        Rp.tmp = 31 - Rp.tmp NEXT   !Convert to request priority
        DECODE last.one(Ext.int.req) LEQ last.one(Soft.int.req) =>
            BEGIN
            Of :=   BEGIN   !Software interrupt
                    Soft.Int.req<Rp.tmp> = 0;
                    trap(SI.vec,1) NEXT
                    Write.ctx(Maxreg+1) = Rp.tmp
                    END,
            1t :=   BEGIN   !External Interrupt
                    Ext.int.req<Rp.tmp> = 0;
                    DECODE Mw[Ext.vec[Rp.tmp]]<0> =>
                        BEGIN
                        1:= trap(Ext.vec[Rp.tmp],0),  !Just vector to it
                        0:= BEGIN   !Parameterized Handler
                        trap(PI.vec,1) NEXT
                        IF Maxreg=> write.ctx(maxreg+1)=Mw[Ext.vec[Rp.tmp]]
                            END
                        END
                    END
            END NEXT
        Pri = Rp.tmp        !Raise priority
    END,
trap(vec<0:31>,nparms<0:7>) :=
    BEGIN
    Write.ctx(0) = PC NEXT
    Write.ctx("FFFFFFFF) = PSW NEXT
    last.mode = Kernel NEXT
    Kernel = 1 NEXT
    build.stack(0,Mw[vec],nparms) NEXT
    Base = 1
    END,
Power.up :=
    BEGIN
    Mer = Mw[Kernel.save] NEXT
    Kctxp = Mw[Mer] AND "FFFFFFFC; Kernel = Mw[Mer]<31>;
    Mptr[1] = Mw[Mer+4];
    Trap(Mw[PR.vec],0) NEXT
    Pri = "1F
    END,
** Instructions ** (TC)
    tmp<31:0>,                      ! global single precision temporaries
    tmp1<31:0>,
    tmp2<31:0>,
    tmp.d<63:0>,                    ! global double precision temporary
    tmp.t<>,                        ! temporary for truncate info
Privilege.chk :=
    BEGIN
```

```
        BEGIN
            write.ctx("FFFFFFFF) = PSW NEXT
            write.ctx(0) = pc.back NEXT
            superv = 1 NEXT
            build.stack(0,Mw[priv.error],1) NEXT
            write.ctx(maxreg+1) = Memory@Byte@pc.back<31>@pc.back<2 6:0>
        END
    END,
Memory.Chk :=
    BEGIN
    IF op.addr<Op.Type> NEQ Memory => Exception(Ill.addr)
    END,
! integer add
ADD.ex:=
    BEGIN
    tmp = get.int(single) NEXT                !get first operand
    c@put.int(single,replace) = get.int(single)+tmp
    END,
! integer subtract
SUB.ex:=
    BEGIN
    tmp = get.int(single) NEXT  !get minuend
    c@put.int(single,replace) = get.int(single) + NOT tmp +(US) 1
    END,
! integer multiply single precision
MUL.ex:=
    BEGIN
    tmp = get.int(single) NEXT                ! get first operand
    put.int(single,replace) = get.int(single)*tmp
    END,
! integer divide single precision without remainder
DIV.ex:=
    BEGIN
    tmp = get.int(single) NEXT                ! get first operand
    IF tmp EQL 0 => Exception(Ill.Divisor) NEXT
    put.int(single,replace) = get.int(single)/tmp
    END,
! integer negate
NEG.ex:=
    BEGIN
    c@put.int(single,replace) = NOT get.int(single) +(US) 1
    END,
! logical NOT
NOT.ex:=
    BEGIN
    put.log(replace) = NOT get.log()
    END,
! integer remainder B over A
REM.ex:=
    BEGIN
    tmp1    = get.int(single) NEXT
    IF tmp1 EQL 0 => Exception(Ill.Divisor) NEXT
    put.int(single,0) = get.int(single) MOD tmp1
```

```
   END,
! integer modulus
MOD.ex:=
   BEGIN
   tmp1    = get.int(single) NEXT
   IF tmp1 EQL 0 => Exception(Ill.Divisor) NEXT
   put.int = get.int(single) MOD tmp1 NEXT
   IF get.int<31> XOR tmp1<31> => put.int = put.int + tmp1 NEXT
   put.int(single,0)
   END,
! integer multiply double precision
EMUL.ex:=                          ! extended integer multiply
   BEGIN
   tmp = get.int(single) NEXT
   put.int(double,0) = get.int(single)*tmp
   END,
! integer divide double precision with remainder
EDIV.ex:=                  ! extended integer divide (with remainder)
   BEGIN
   tmp = get.int(single) NEXT              ! get divisor
   IF tmp EQL 0 => Exception(Ill.Divisor) NEXT
   tmp.d = get.int(double) NEXT
   put.int(single,0) = tmp.d MOD tmp NEXT  ! compute rem(B/A)
   tmp.t = t NEXT
   put.int(double,0) = tmp.d/tmp NEXT              ! compute B/A
   t = t OR tmp.t OR ( tmp.d<63> AND tmp<31> AND put.int<63>)
   END,
! integer increment and decrement by fixed constants
!          R = R + nnn
!          R = R - nnn
INC.ex :=  (put.int(single,1) = get.int(single) + 1),      ! increment by 1
INC2.ex := (put.int(single,1) = get.int(single) + 2),      ! increment by 2
INC4.ex := (put.int(single,1) = get.int(single) + 4),      ! increment by 4
INC8.ex := (put.int(single,1) = get.int(single) + 8),
DEC.ex :=  (put.int(single,1) = get.int(single) + "FFFFFFFF),
! integer add single precision with carry in
ADDC.ex:=                          ! R = B + A + carry
   BEGIN
   tmp = get.int(single) NEXT
   c@put.int(single,0) = get.int(single) + tmp + (US)c
   END,
! integer subtract single precision with carry in
SUBC.ex:=                          ! R = B + (NOT A) + carry
   BEGIN
   tmp = get.int(single) NEXT      ! get  A
   c@put.int(single,0) = get.int(single) + (NOT tmp) + (us)c
   END,
! sign extended move
MOV.ex:= (put.int(single,0) = get.int(single)),
! integer compare A with B
CMP.ex:=                           ! integer compare A with B
   BEGIN
   tmp1    = get.int(single) NEXT
   tmp2    = get.int(single) NEXT
```

```
    z = tmp1 EQL tmp2;
    n = tmp1 LSS tmp2;
    t = 0
    END,
! integer compare within bounds
CMPWB.ex:=                  ! compare integer tmp with bounds A and B
    BEGIN
    tmp = get.int(single) NEXT              ! get variable
    tmp1    = get.int(single) NEXT          ! get 1st bound
    tmp2    = get.int(single) NEXT          ! get 2nd bound
    z = tmp GEQ tmp1  AND tmp LEQ tmp2;
    n = tmp LSS tmp1;
    t = 0
    END,
! range check
RANGE.ex :=
    BEGIN
    tmp = get.int(single) NEXT              ! get variable
    tmp1    = get.int(single) NEXT          ! get 1st bound
    tmp2    = get.int(single) NEXT          ! get 2nd bound
    IF tmp LSS tmp1 OR tmp GTR tmp2 =>
        exception(Range.error)
    END,
! integer compare A with ZERO
TEST.ex:=                           ! compare A with 0,
    BEGIN
    get.int(single) NEXT
    z = get.int EQL 0;
    n = get.int LSS 0;
    t = 0
    END,
ABS.ex:=            !Absolute value
    BEGIN
    IF get.int(single)<0> => get.int = NOT get.int + 1 NEXT
    put.int(single,0) = get.int
    END,
EQL.ex:=
    BEGIN
    put.log(0) <= z
    END,
NEQ.ex:=
    BEGIN
    put.log(0) <= NOT z
    END,
LSS.ex:=
    BEGIN
    put.log(0) <= n
    END,
GTR.ex:=
    BEGIN
    put.log(0) <= NOT (n OR z)
    END,
LEQ.ex:=
    BEGIN
```

```
   put.log(0) <= n OR z
   END,
GEQ.ex:=
   BEGIN
   put.log(0) <= NOT n
   END,
! arithmetic shift left and right single precision
ASH.ex:=                              ! R= B shifted |A| bit positions
                                      ! R = R shifted |A| bit positions
                                      ! if A >= 0:  LEFT shift with zero fill
                                      ! if A < 0:  RIGHT shift with sign fill
                                     .! t = 1 if sign changes during shift
   BEGIN
   tmp1 = get.int(single) NEXT              ! get shift specifier
   tmp2 = get.int(single) NEXT              ! get source  sign extend
   DECODE tmp1<31:5> =>
           BEGIN
           "0000000 :=     BEGIN
                           put.int(single,0) <= tmp2 SLO tmp1 NEXT
                           t = t OR tmp2 NEQ (put.int SRD tmp1)
                           END,
           "7FFFFFF :=     BEGIN
                           put.int(single,0) = tmp2 SRD (NOT tmp1 + 1)
                           END,
           OTHERWISE := DECODE tmp1<31> =>
                           BEGIN
                           0:=     BEGIN
                                   put.int(single,0) = 0;
                                   t = tmp2 NEQ 0
                                   END,
                           1:=     BEGIN
                                   put.int(single,0) <= tmp2<31>;
                                   t = 0
                                   END
                           END
           END
   END,
! logical AND single precision
AND.ex:=
   BEGIN
   tmp = get.log() NEXT
   put.log(replace) = get.log() AND tmp
   END,
! logical OR single precision
OR.ex:=
   BEGIN
   tmp = get.log() NEXT
   put.log(replace) = get.log() OR tmp
   END,
! logical XOR single precision
XOR.ex:=
   BEGIN
   tmp = get.log() NEXT
   put.log(0) = get.log() XOR tmp
```

```
   END,
! rotate
ROT.ex :=
   BEGIN
   tmp1 = get.int(single) NEXT
   tmp2 = get.int(single) NEXT
   DECODE tmp1<31> =>
       BEGIN
       DECODE op.addr<op.size> =>
           BEGIN
           Dwrd := put.log(0) = UNDEFINED(),
           Byte := put.log(0) = tmp2<7:0> SLR tmp1<4:0>,
           Hwrd := put.log(0) = tmp2<15:0> SLR tmp1<4:0>,
           Word := put.log(0) = tmp2<31:0> SLR tmp1<4:0>
           END,
       DECODE op.addr<op.size> =>
           BEGIN
           Dwrd := put.log(0) = UNDEFINED(),
           Byte := put.log(0) = tmp2<7:0> SRR (-tmp1)<4:0>,
           Hwrd := put.log(0) = tmp2<15:0> SRR (-tmp1)<4:0>,
           Word := put.log(0) = tmp2<31:0> SRR (-tmp1)<4:0>
           END
       END
   END,
! Logical Shift
LSH.ex :=
   BEGIN
   tmp1 = get.int(single) NEXT
   tmp2 = get.log( ) NEXT
   DECODE tmp1<31:5> =>
           BEGIN
           "0000000 :=     put.log(0) = tmp2 SLO tmp1,
           "7FFFFFF :=     put.log(0) = tmp2 SRO (NOT tmp1 + 1),
           OTHERWISE :=    put.log(0) = 0
           END
   END,
! logical move
MOVL.ex:= (put.log(0) = get.log()),
EXCH.ex :=
   BEGIN
   tmp = get.log() NEXT            !get A
   tmp.d = op.addr NEXT            !save location of A
   put.log(1) = get.log() NEXT     !get B and insure writable
   store(tmp.d) = get.log NEXT     !write B into A
   put.log(1) = tmp                !Write A into B
   END,
MOVA.ex:=
   BEGIN
   op.addr() NEXT
   Memory.Chk( ) NEXT
   put.log(0) = op.addr
   END,
MOVBK.ex:= !Move Block
   BEGIN
```

```
    tmp = get.int(single) NEXT
    op.addr() NEXT
    Memory.Chk() NEXT
    tmp.d = op.addr<31:0> NEXT
    op.addr() NEXT
    Memory.Chk() NEXT
    DECODE tmp.d<31:0> GTR(US) op.addr<31:0> =>
        BEGIN
    0 :=    BEGIN           ! start from the end
            tmp.d<31:0> = tmp.d<31:0> +(US) tmp;
            op.addr<31:0> = op.addr<31:0> +(US) tmp NEXT
            Repeat  BEGIN
                IF tmp EQL(US) 0 => LEAVE movbk.ex NEXT
                store(op.addr) = read(tmp.d<31:0>,op.addr<op.size>,M.r) NEXT
                tmp = tmp -(US) 1;
                tmp.d = tmp.d -(US) 1;
                op.addr<31:0> = op.addr<31:0> -(US) 1
                END !movbkb.loop
            END,    !decode case 0
    1 :=    Repeat  BEGIN   ! normal direction
                IF tmp EQL(US) 0 => LEAVE movbk.ex NEXT
                store(op.addr) = read(tmp.d<31:0>,op.addr<op.size>,M.r) NEXT
                tmp = tmp -(US) 1;
                tmp.d = tmp.d +(US) 1;
                op.addr<31:0> = op.addr<31:0> +(US) 1
                END,
        END !decode
    END,
MOVM.ex:=
    BEGIN
    tmp = get.int(single) NEXT
    get.log() NEXT
    op.addr() NEXT
    Memory.Chk() NEXT
    Repeat  BEGIN
            IF tmp EQL 0 => LEAVE movm.ex NEXT
            store(op.addr) = get.log NEXT
            tmp = tmp -(US) 1;
            op.addr<31:0> = op.addr<31:0> +(US) 1
            END
    END,
! logical compare zero extended
CMPU.ex:=
    BEGIN
    tmp1 = get.log() NEXT
    tmp2 = get.log() NEXT
    z = tmp1 EQL tmp2;                          ! A=B
    n = tmp1 LSS(US) tmp2;          ! A<B
    t = 0
    END,
! clear operand and condition codes
CLR.ex:=
    BEGIN
    put.log(0) = 0                  ! clear operand
```

```
    END,
JUMP.ex:=
    BEGIN
    op.addr() NEXT
    Memory.Chk() NEXT
    PC = op.addr
    END,
!=============================================================================
!  Floating Point Instructions
REQUIRE.ISP | FLOAT.ISP | ,    !Floating arithmetic operators
          ! floating add
addf.ex :=
    BEGIN
        ftmp = get.float() NEXT
        fact = get.float() NEXT
        float.add() NEXT
        put.float(replace) = ftmp
    END,
          ! floating subtract
subf.ex :=
    BEGIN
        fact = get.float() NEXT
        ftmp = get.float() NEXT
        fact<s> = NOT fact<s> NEXT
        float.add() NEXT
        put.float(replace) = ftmp
    END,
!  Floating Multiply
MULF.ex  :=
  BEGIN
  ftmp = get.float() NEXT
  fact = get.float() NEXT
  float.mult() NEXT
  put.float(replace) = ftmp
  END,
!  Floating Divide
DIVF.ex  :=
  BEGIN
  fact = get.float() NEXT
  ftmp = get.float() NEXT
  float.div() NEXT
  put.float(replace) = ftmp
  END,
!  Negate Floating
NEGF.ex  :=
  BEGIN
  get.float() NEXT
  put.float(replace) = NOT get.float<70> @ get.float<69:0>
  END,
!  Convert integer to floating
FLOAT.ex  :=
  BEGIN
  tmp1 = get.int(0) NEXT
  put.float(0) = int2float(tmp1)
```

```
  END,
!  Convert floating to integer
FIX.ex  :=
  BEGIN
  fact = get.float() NEXT
  DECODE (fact<e> EQL "7FF) AND (fact<f> NEQ 0) =>
    BEGIN
    0  :=
      BEGIN
      t = 0 NEXT
      float2int(fact) NEXT
      DECODE t  =>
   BEGIN
   0  := put.int(single,0) = float2int,
   1  :=
     BEGIN
     put.int(single,0) = float2int NEXT
     t = 1
     END
   END
      END,
    1  :=
      BEGIN
      put.int(single,1) = get.int(single) NEXT
      fp.exception(Ill.Operation)
      END
    END
  END,
!  Move floating
MOVF.ex  :=
  BEGIN
  put.float(0) = get.float()
  END,
!  Clear floating
CLRF.ex  :=
  BEGIN
  put.float(0) = 0
  END,
!  Compare floating
CMPF.ex  :=
  BEGIN
  ftmp = get.float() NEXT
  fact = get.float() NEXT
  float.cmp()
  END,
!  Square Root floating
SQRTF.ex  :=
  BEGIN
  fact = get.float() NEXT
  put.float(0) = fp.sqrt(fact)
  END,
!  Absolute value floating
ABSF.ex  :=
  BEGIN
```

```
  put.float(0) = 0 @ get.float()<69:0>
  END,
!  Round floating to integer value
RNDI.ex  :=
  BEGIN
  fact = get.float() NEXT
  put.float(0) = rnd2int(fact)
  END,
!  Floating Remainder
REMF.ex  :=
  BEGIN
  fact = get.float() NEXT
  ftmp = get.float() NEXT
  fp.rem() NEXT
  put.float(0) = ftmp
  END,
!=============================================================================
branch(condition<>)<1:0> :=                          !common branch routine
BEGIN
branch = long.branch +(US) 1 NEXT  !Size of displacement
DECODE condition =>
   BEGIN
   pc = pc + size(branch),          !NO branch
   pc = pc + sxt(read(pc,branch,M.i),branch)
   END
END,
!Unconditional Branch
BR.ex := (branch(1)),
!Branch on equal
BEQL.ex := (branch(z)),
!Branch not equal
BNEQ.ex := (branch(NOT z)),
!Branch less or equal
BLEQ.ex := (branch( z OR n)),
!Branch on less
BLSS.ex := (branch(n)),
!Branch greater or equal
BGEQ.ex := (branch(NOT n)),
!Branch greater than
BGTR.ex := (branch(NOT (n or z))),
!Branch if carry set
BCS.ex := (branch(c)),
!Branch if carry clear
BCC.ex := (branch(NOT c)),
!Branch if truncate set
BTS.ex := (branch(t)),
!Branch if truncate clear
BTC.ex := (branch(NOT t)),
CASE.ex:=
   BEGIN
   tmp = get.int(single) NEXT
   tmp = tmp - get.int(single) NEXT
   DECODE tmp LSS(US) get.int(single) =>
          BEGIN
```

```
            pc = pc + (get.int * 2),        !Sel exceeds Num - 1
            pc = pc + sxt(read(pc + (tmp * 2),Hwrd,M.i),Hwrd)
            END
      END,
   LOOP.ex:=
      BEGIN
      tmp = get.int(single) NEXT      !get increment
      put.int(single,1) = get.int(single) + tmp NEXT  !add to counter
      get.int(single) NEXT    !get limit
      long.branch = 1 NEXT
      branch(((tmp GEQ 0) AND (put.int LEQ get.int)) OR
            ((tmp LSS 0) AND (put.int GEQ get.int)))
      END,
   ENTLP.ex:=
      BEGIN
      tmp = get.int(single) NEXT      !get initial counter
      pc = pc + sxt(read(pc,Hwrd,M.i),Hwrd) NEXT  !get disp to loop control
      DECODE read(pc,byte,M.i) =>
            BEGIN
            "27:=   BEGIN                           ! LOOP
                  pc = pc + 1 NEXT
                  get.int(single) NEXT    !get increment
                  store(op.addr()) = tmp NEXT     !load counter
                  tmp = get.int NEXT      !save increment
                  get.int(single) NEXT    !get limit
                  long.branch = 1 NEXT
                  branch(((tmp GEQ 0) AND (store LEQ get.int)) OR
                        ((tmp LSS 0) AND (store GEQ get.int)))
                  END,
            "2D:=   BEGIN                           ! IBLEQ
                  pc = pc + 1 NEXT
                  store(op.addr()) = tmp NEXT     !load counter
                  get.int(single) NEXT            !get limit
                  long.branch = 1 NEXT
                  branch(store LEQ get.int)
                  END,
            "29:=   BEGIN                           ! IBLSS
                  pc = pc + 1 NEXT
                  store(op.addr()) = tmp NEXT     !load counter
                  get.int(single) NEXT            !get limit
                  long.branch = 1 NEXT
                  branch(store LSS get.int)
                  END,
            "2B:=   BEGIN                           ! DBGEQ
                  pc = pc + 1 NEXT
                  store(op.addr()) = tmp NEXT     !load counter
                  get.int(single) NEXT            !get limit
                  long.branch = 1 NEXT
                  branch(store GEQ get.int)
                  END,
            "2F:=   BEGIN                           ! DBGTR
                  pc = pc + 1 NEXT
                  store(op.addr()) = tmp NEXT     !load counter
                  get.int(single) NEXT            !get limit
```

```
                    long.branch = 1 NEXT
                    branch(store GTR get.int)
                    END,
            OTHERWISE:= (exception(Bad.Displacement))
            END !of DECODE
    END,
 IBLEQ.ex:=
    BEGIN
    store(op.addr) = get.int(single) + 1 NEXT       !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store LEQ get.int)
    END,
 IBLSS.ex:=
    BEGIN
    store(op.addr) = get.int(single) + 1 NEXT       !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store LSS get.int)
    END,
 DBGEQ.ex:=
    BEGIN
    store(op.addr) = get.int(single) - 1 NEXT       !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store GEQ get.int)
    END,
 DBGTR.ex:=
    BEGIN
    store(op.addr) = get.int(single) - 1 NEXT       !increment counter
    get.int(single) NEXT                !get limit
    long.branch = 1 NEXT
    branch(store GTR get.int)
    END,
 CALL.ex :=          !Procedure call
    BEGIN
    op.addr() NEXT
    op.addr<30:31> = 0;
    Memory.Chk() NEXT
    call.switch(op.addr) NEXT
    IF ir<0> => Privilege = 0
    END,
 SVC.ex :=
    BEGIN
    get.log() NEXT
    vector.call(1,get.log)
    END,
 JSR.ex:=
    BEGIN
    op.addr() NEXT
    Memory.Chk() NEXT
    Write.Ctx(1) = Read.Ctx(1) - 4 NEXT     !SP <- SP-4
    write(Write.Ctx,Word,M.w) = pc NEXT
    pc = op.addr<31:0>
```

```
    END,
RSR.ex:=
    BEGIN
    pc = read(Read.Ctx(1),Word,M.r) NEXT
    Write.Ctx(1) = Read.Ctx + 4
    END,
RET.ex :=
    BEGIN
    call.restore()
    END,
ERET.ex :=
    BEGIN
    UDLE = 0; Write.ctx(maxreg+MaxPar+1) = 0 NEXT
    get.log() NEXT
    Exception(get.log)
    END,
ERP.ex :=
    BEGIN
    DECODE Base =>
        BEGIN
            BEGIN   !There is a caller
                Get.log() NEXT
                Call.restore() NEXT
                Exception(get.log) = 1      !force to user handler
            END,
            exception(Context.base) !no caller
        END
    END,
RAISE.ex:=
    BEGIN
    exception(get.log())
    END,
ECODE.ex :=
    BEGIN
    Put.log(0) = Read.ctx(maxreg+MaxPar+1) NEXT
    Write.ctx(maxreg+MaxPar+1) = 0
    END,
EXCEPT.ex :=
    BEGIN
    op.addr() NEXT
    Memory.Chk() NEXT
    Write.ctx(maxreg+MaxPar+1) = op.addr
    END,
LPSW.ex :=
    BEGIN
    Privilege.Chk() Next
    IF Base => exception(Context.Base) NEXT
    Write.ctx(Maxreg+Maxpar+3) = get.log()
    END,
SPSW.ex :=
    BEGIN
    Privilege.Chk() Next
    IF Base => exception(Context.Base) NEXT
    put.log(0) = Read.ctx(Maxreg+Maxpar+3)
```

```
    END,
BREAK.ex :=
    BEGIN
    Sup.eh(0,Break)
    END,
NOP.ex:=
    BEGIN
    NO.OP()
    END,
LTASK.ex :=
    BEGIN
    Privilege.Chk() Next
    Get.int(double) NEXT
    IF get.int<33:32> => exception(Context.Alignment);
    IF get.int<1:0> EQL '10 => exception(spec.error) NEXT
    Tctxp = get.int<63:32>;
    Mptr[0] = get.int<31:0>
    END,
STASK.ex :=
    BEGIN
    Privilege.Chk() Next
    put.int(double,0) = Tctxp@Mptr[0]
    !Flush task context to memory
    END,
TSTART.ex :=
    BEGIN
    Privilege.Chk() Next
    Write.ctx("FFFFFFFF) = PSW NEXT
    Write.ctx(0) = PC NEXT
    IF get.log()<31:1> NEQ 0 => exception(spec.error) NEXT
    IF ir<0> => pop.stack() NEXT
    Kernel = get.log<0> NEXT
    PC = read.ctx(0) NEXT
    PSW = read.ctx("FFFFFFFF)
    END,
TRAISE.ex :=
    BEGIN
    get.log() NEXT
    Tstart.ex() NEXT
    PC.back = pc NEXT
    Exception(get.log)
    END,
TINIT.ex :=
    BEGIN
    Privilege.Chk() Next
    Write.ctx("FFFFFFFF) = PSW NEXT
    Write.ctx(0) = PC NEXT
    PC = get.log() NEXT
    get.log() NEXT  !get process half of psw
    IF ir<0> => pop.stack() NEXT
    PSW<0:15> = get.log;
    Kernel = NOT Kernel NEXT
    Build.stack(0,PC,0) NEXT
    Base = 1
```

```
   END,
SBF.ex:=
   BEGIN
   tmp = get.log( ) NEXT    !get source
   tmp1 = get.int(single) NEXT     !get position
   put.field(tmp1,get.log( ),0) = tmp
   END,
LBFS.ex:=          !Load Bit Field (Sign extended)
   BEGIN(US)
   tmp = get.int(single) NEXT      !get position
   get.field(tmp,get.log()) NEXT
   put.int(single,0) = ((get.field SLO (32- get.log)) SRD (32- get.log))
   END,
LBF.ex:=           !Load Bit Field
   BEGIN
   tmp = get.int(single) NEXT      !get position
   put.log(0) = get.field(tmp,get.log())
   END,
SETBIT.ex:=
   BEGIN
   !Operation is interlocked (read-modify-write)
   n = get.field(get.int(single),1) NEXT
   put.field(get.int,1,0) = 1 NEXT
   z = NOT n;
   t = 0
   END,
CLRBIT.ex:=
   BEGIN
   !Operation is interlocked (read-modify-write)
   n = get.field(get.int(single),1) NEXT
   put.field(get.int,1,0) = 0 NEXT
   z = NOT n;
   t = 0
   END,
INVBIT.ex:=
   BEGIN
   !Operation is interlocked (read-modify-write)
   n = get.field(get.int(single),1) NEXT
   put.field(get.int,1,0) = z = NOT n;
   t = 0
   END,
TSTBIT.ex:=                !Test Bit
   BEGIN
   n = get.field(get.int(single),1)<0> NEXT
   z = NOT n;
   t = 0
   END,
PUSH.ex:=          !push onto sp stack
   BEGIN
   IF maxreg EQL 0 => exception(Ill.reg) NEXT
   write.ctx(1) = read.ctx(1) - 4 NEXT
   write(write.ctx,word,M.w) = get.log( )
   END,
POP.ex:=           !pop from sp stack
```

```
   BEGIN
   IF maxreg EQL 0 => exception(Ill.reg) NEXT
   put.log(0) = read(read.ctx(1),word,M.r) NEXT
   write.ctx(1) = read.ctx + 4
   END,
MTS.ex:=          !Move To Stack
   BEGIN
   tmp1 = get.log() NEXT
   tmp = op.addr<op.size> NEXT
   store = get.log() - size(tmp) NEXT      !compute new S but don't store
   write(store,tmp,M.w) = tmp1 NEXT
   store(op.addr)          !store S now
   END,
MFS.ex:=          !Move From Stack
   BEGIN
   tmp = get.log() NEXT
   tmp.d = op.addr NEXT    !save addr of S
   tmp1 = op.addr()<op.size> NEXT  !get size A
   put.log(1) = read(tmp,tmp1,M.r) NEXT    !store A
   store(tmp.d) = tmp + size(tmp1)         !store new S
   END,
ILIST.ex:=        !Insert in doubly linked LIST
   BEGIN
   tmp = op.addr() NEXT    !get entry address (E)
   Memory.Chk() NEXT
   tmp1 = op.addr() NEXT   !get address of entry to insert after (P)
   Memory.Chk() NEXT
   !get address of successor of P (S) and check for write rights
   tmp2 = read(tmp1,word,M.w) NEXT
   read(tmp,word,M.w) NEXT         !check write rights
   read(tmp+4,word,M.w) NEXT       !check write rights
   read(tmp2+4,word,M.w) NEXT      !check write rights
   !IF tmp<1:0> OR tmp1<1:0> OR tmp2<1:0> => exception(Ill.Operand) NEXT
   write(tmp,word,M.w) = tmp2 NEXT !E(fwd) <= S
   write(tmp+4,word,M.w) = tmp1 NEXT       !E(back) <= P
   write(tmp2+4,word,M.w) = tmp NEXT       !S(back) <= E
   write(tmp1,word,M.w) = tmp      !P(fwd) <= E
   END,
RLIST.ex:=        !Remove from doubly linked LIST
   BEGIN
   tmp = op.addr() NEXT    !get entry address E
   Memory.Chk() NEXT
   tmp1 = read(tmp,word,M.r) NEXT  !get address of successor (S)
   tmp2 = read(tmp+4,word,M.r) NEXT        !get address of pred (P)
   read(tmp2,word,M.w) NEXT        !check access rights
   read(tmp1+4,word,M.w) NEXT      !check access rights
   !IF tmp<1:0> OR tmp1<1:0> OR tmp2<1:0> => exception(Ill.Operand) NEXT
   write(tmp2,word,M.w) = tmp1 NEXT        !P(fwd) <= S
   write(tmp1+4,word,M.w) = tmp2 NEXT      !S(back) <= P
   store(op.addr()) = tmp
   END,
MULFIX.ex:=               !Multiply fixed point
   BEGIN(TC)
   tmp = 0 NEXT
```

```
      tmp1 = get.int(single) NEXT
      IF tmp1<31> => (tmp<0> = 1; tmp1 = -tmp1) NEXT
      put.int = get.int(single) NEXT
      IF put.int<63> => (tmp<1> = 1; put.int = -put.int) NEXT
      put.int = put.int * get.int NEXT
      DECODE get.int(single)<31> =>
              BEGIN
              put.int = put.int SRD get.int,
              put.int = put.int SLO (NOT get.int + 1)
              END NEXT
      put.int(single,0) = put.int * tmp<0> * tmp<1>
      END,
   DIVFIX.ex:=                !Divide fixed point
      BEGIN(TC)
      tmp = 0 NEXT
      tmp1 = get.int(single) NEXT
      IF tmp1 EQL 0 => exception(Ill.Divisor) NEXT
      IF tmp1<31> => (tmp<0> = 1; tmp1 = -tmp1) NEXT
      tmp.d = get.int(single) NEXT
      IF tmp.d<63> => (tmp<1> = 1; tmp.d = -tmp.d) NEXT
      DECODE get.int(single)<31> =>
              BEGIN
              tmp.d = tmp.d SLO get.int,
              tmp.d = tmp.d SRD (NOT get.int + 1)
              END NEXT
      put.int(single,0) = (tmp.d / tmp1) * tmp<0> * tmp<1>
      END,
   CMPS.ex:=  !Compare and Swap
      BEGIN
      tmp = get.log() NEXT
      !Serialization and Memory Lock
      tmp1 = get.log() NEXT
      tmp2 = op.addr NEXT     !Save location of second operand
      DECODE z = (tmp1 EQL get.log()) =>
              BEGIN
              (store(op.addr) = tmp1; n = tmp1 LSS(US) get.log),
              (store(tmp2) = tmp; n = 0)
              END
      !memory unlock
      END,
   SIZE.ex:=
      BEGIN
      op.addr() NEXT
      put.log(0) = size(op.addr<op.size>)
      END,
   SETCC.ex:=          !SET Condition Codes
      BEGIN
      eae@c@t@n@z = get.log()
      END,
   REPENT.ex:=                !REPlace ENTry in map
      BEGIN
      Privilege.chk() NEXT
      tmp.d = get.int(1) NEXT !Map Entry
      tmp1 = get.log() NEXT   !Map number
```

```
   IF tmp1 GTR 1 => exception(Spec.Error) NEXT
   IF Mw[(Mptr[tmp1]<Map.addr>@'000)-4] LSS(US) get.log() =>
          exception(Spec.error) NEXT      !Check map size(from memory)
   Md[(Mptr[tmp1]<Map.addr> + get.log)@'000] = tmp.d
   !Invalidate any translation buffer associated with this entry
   !Update any copies of the map size
   END,
MAP.ex:=            !Map virtual address
   BEGIN
   Privilege.chk() NEXT
   get.log() NEXT  !address
   vp(get.log,M.n) NEXT    !translate address
   put.log(0) = vp NEXT
   put.log(0) = seg.number
   END,
WINDOW.ex:=                 !window to micromachine (Implementation
 dependent)
   BEGIN
   !IF console.enabled => Break to microcode NEXT
   PC = PC + 1;
   Stop()
   END,
ENI                         !Nebula ISP description
```


N
DAT
ILM

```
      put.log(1) = tmp                    !Write A into B
      END,
   MOVA.ex:=
      BEGIN
      op.addr() NEXT
      Memory.Chk() NEXT
      put.log(0) = op.addr
      END,
   MOVBK.ex:= !Move Block
      BEGIN
```

```
      BEGIN
      tmp1 = get.log() NEXT
      tmp2 = get.log() NEXT
      z = tmp1 EQL tmp2;                    ! A=B
      n = tmp1 LSS(US) tmp2;                ! A<B
      t = 0
      END,
   ! clear operand and condition codes
   CLR.ex:=
      BEGIN
      put.log(0) = 0                        ! clear operand
```

```
    BEGIN
    get.float() NEXT
    put.float(replace) = NOT get.float<70> @ get.float<69:0>
    END,
!   Convert integer to floating
FLOAT.ex  :=
    BEGIN
    tmp1 = get.int(0) NEXT
    put.float(0) = int2float(tmp1)
```

```
    float.cmp()
    END,
!  Square Root floating
SQRTF.ex  :=
    BEGIN
    fact = get.float() NEXT
    put.float(0) = fp.sqrt(fact)
    END,
!  Absolute value floating
ABSF.ex  :=
    BEGIN
```

```
!Branch if truncate set
BTS.ex := (branch(t)),
!Branch if truncate clear
BTC.ex := (branch(NOT t)),
CASE.ex:=
   BEGIN
   tmp = get.int(single) NEXT
   tmp = tmp - get.int(single) NEXT
   DECODE tmp LSS(US) get.int(single) =>
           BEGIN
```

```
                    pc = pc + 1 NEXT
                    store(op.addr()) = tmp NEXT      !load counter
                    get.int(single) NEXT             !get limit
                    long.branch = 1 NEXT
                    branch(store GEQ get.int)
                    END,
"2F:=               BEGIN                            ! DBGTR
                    pc = pc + 1 NEXT
                    store(op.addr()) = tmp NEXT      !load counter
                    get.int(single) NEXT             !get limit
```

```
    vector.call(1,get.log)
    END,
JSR.ex:=
    BEGIN
    op.addr() NEXT
    Memory.Chk() NEXT
    Write.Ctx(1) = Read.Ctx(1) - 4 NEXT      !SP <- SP-4
    write(Write.Ctx,Word,M.w) = pc NEXT
    pc = op.addr<31:0>
```

```
LPSW.ex :=
   BEGIN
   Privilege.Chk() Next
   IF Base => exception(Context.Base) NEXT
   Write.ctx(Maxreg+Maxpar+3) = get.log()
   END,
SPSW.ex :=
   BEGIN
   Privilege.Chk() Next
   IF Base => exception(Context.Base) NEXT
   put.log(0) = Read.ctx(Maxreg+Maxpar+3)
```

```
Privilege.Chk() Next
Write.ctx("FFFFFFFF) = PSW NEXT
Write.ctx(0) = PC NEXT
PC = get.log() NEXT
get.log() NEXT  !get process half of psw
IF ir<0> => pop.stack() NEXT
PSW<0:15> = get.log;
Kernel = NOT Kernel NEXT
Build.stack(0,PC,0) NEXT
Base = 1
```