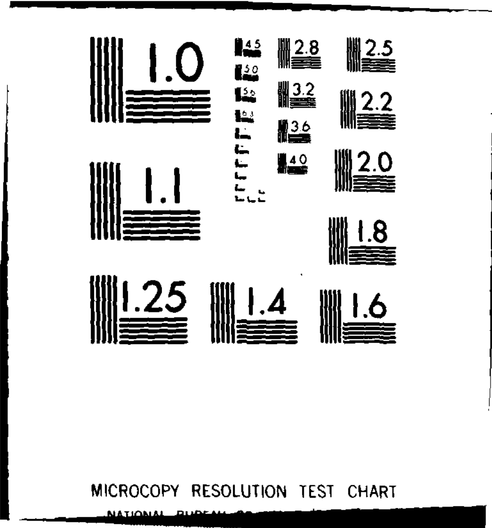
AD-A093 735 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
DEVELOPMENT OF A METHODOLOGY FOR VERIFYING MILITARY COMPUTER FA--ETC(U)
SEP 80 J B CLARY, R K JOOBBANI, F M SMITH DAAK80-79-C-0780
UNCLASSIFIED RTI/1822/00-01F CORADCOM-80-0780-F NL

1 OF 2
AD A
093735


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

LEVEL II

(12)

RESEARCH AND DEVELOPMENT TECHNICAL REPORT
CORADCOM- 80-0780-F

AD A093735

DEVELOPMENT OF A METHODOLOGY FOR VERIFYING MILITARY COMPUTER FAMILY BUILT-IN-TEST PERFORMANCE SPECIFICATIONS

J. B. Clary
R. K. Joobbani
F. M. Smith
RESEARCH TRIANGLE INSTITUTE
P.O. Box 12194
Research Triangle Park, NC 27709

DTIC
ELECTE
JAN 13 1981
S D
E

September 1980

Final Report for Period 22 May 1979 to 21 May 1980

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

Prepared for:

CENTER FOR TACTICAL COMPUTER SYSTEMS

FILE COPY

CORADCOM

US ARMY COMMUNICATIONS RESEARCH & DEVELOPMENT COMMAND
FORT MONMOUTH, NEW JERSEY 07703

81 1 12 067

NOTICES

Disclaimers

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: CORADCOM-80-0780-F
2. GOVT ACCESSION NO.: AD-A093735
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): Development of a Methodology for Verifying Military Computer Family Built-In-Test Performance Specifications.
5. TYPE OF REPORT & PERIOD COVERED: Final Report, May 1979 – May 1980
6. PERFORMING ORG. REPORT NUMBER: RTI/1822/00-01F
7. AUTHOR(s): J. B. Clary, R. K. Joobbani, F. M. Smith
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: Tasks 1.0 and 2.0
11. CONTROLLING OFFICE NAME AND ADDRESS: (CENTACS) Test, Measurement & Diagnostic Systems Division, U. S. Army Communications Research & Devel. Comm., Ft. Monmouth, NJ 07703
12. REPORT DATE: June 1980 — Sep 80
13. NUMBER OF PAGES: 179
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Test, Measurement & Diagnostic Systems Division, U. S. Army Communications Research & Devel. Comm., Ft. Monmouth, NJ 07703 — 183
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:
16. DISTRIBUTION STATEMENT (of this Report): Approved for Public Release; Distribution Unlimited
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): Fault Detection, Fault Isolation, Built-In-Test, Self-Testing Computers

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

A previous study has addressed the identification of built-in-test (BIT) techniques for the Military Computer Family (MCF). In that study, appropriate BIT techniques were identified based upon an assumed fault population. A model was developed to predict where in the system these faults are most likely to occur. Based upon this model, a rationale was developed for deploying built-in-test resources and a unified BIT approach for MCF was recommended. A later study focused on the identification of relevant BIT requirements for Military Computer Family form, fit and function (F^3) specifications.

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

411039

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The present study was concerned with extending the BIT performance assessment methodology for verifying MCF built-in-test performance and the application of this methodology to a particular member of the Military Computer Family.

The recommended approach is to describe MCF functional modules using ISP language descriptions. The PDP-11/70 member was selected as representative. The PDP-11/70 modules were verified to insure their proper functional behavior and the previously recommended BIT approaches were described in ISP and applied to these modules.

In order to validate BIT performance for modules, a functional fault model was developed. The model developed in this study started with basic fault occurrences at the logic level and related these faults to their functional-level manifestations. This fault model was then exercised and the performance of the selected BIT approaches assessed.

PREFACE.

This report was prepared for the Test, Measurement and Diagnostic Systems (TMDS) Division of the Center for Tactical Computer Systems (CENTACS) under the U. S. Army Communications Research and Development Command (CORADCOM) under the technical direction of Messrs. Richard A. Sacane and George Burbank of TMDS. This report was prepared by the Digital Systems Section of the Systems Engineering Department of the Research Triangle Institute. The Project Leader was Mr. J. B. Clary. Primary contributors to this effort were Mr. F. M. Smith and Mr. R. K. Joobbani.

The authors wish to express their sincere appreciation to Professors Mario Barbacci and Dan Siewiorek of Carnegie-Mellon University for their continuing support. Professor Barbacci supported the current effort both directly and indirectly by making the latest version of the Instruction Set Processor (ISP) descriptive language available. Dr. Siewiorek continues to encourage our efforts in this area and has supplied invaluable advice and technical support.

Finally, the authors wish to thank Mr. Duane Northcutt of Carnegie-Mellon for modifying ISP to include a functional fault injection capability.

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

TABLE OF CONTENTS

Page

PREFACE i

LIST OF FIGURES vii

LIST OF TABLES ix

1. INTRODUCTION 1

a. Background 1
b. Present Study Objectives 1
c. Study Approach 1
d. Report Overview 2

2. HIERARCHICAL COMPUTER DESCRIPTIONS USING ISP 3

a. Introduction 3
b. History 3
c. Differences Between the Research Triangle Institute and Carnegie-Mellon University ISP Descriptions of the PDP-11/70 4
d. Single and Dual Processor Models 12

3. FUNCTIONAL FAULT MODELING 31

a. Introduction 31
b. Functional Fault Modeling for the PDP-11 Family 32

(1) Register Decode Function 34
(2) Instruction Decode and Control Function 34
(3) Data Storage Function 36
(4) Data Transfer Function 36
(5) Data Manipulation Function (ALU) 36

(a) Hypotheses 36
(b) Experimental Approach 38
(c) Measurements 43

1. Fault populations 43
2. Effect of gate-level faults on ALU functions 44
3. Frequency of occurrence of different types of functional faults 44
4. Refined functional fault model 54

TABLE OF CONTENTS
(continued)

Page

4. FUNCTIONAL FAULT INJECTION 55

a. Introduction 55
b. Strategy 55

(1) Program Selection 55
(2) Measurements 56

c. Single-CPU Fault Injection 57

(1) Register Decode 58
(2) Instruction Decode 66
(3) Data Storage Function Fault Injection 71
(4) Data Manipulation 76
(5) Results of the Overall Fault Injection 81

d. Dual-CPU Fault Injection 82

(1) Register Decode 83
(2) Instruction Decode 86
(3) Data Storage Function Fault Injection in Dual CPU . . . 89
(4) Data Manipulation Function Fault Injection in Dual CPU 89
(5) Results of the Overall Fault Injection in the Dual CPU 91

5. REFINED FUNCTIONAL FAULT MAPPING 93

a. Introduction 93
b. Functional Fault Simulation Versus Manifestation Simulation 96
c. More Detailed ALU Fault Injection 99
d. Results Compared to Functional-Level Fault Injection in Chapter 4 103

6. RESULTS, SUMMARY, AND FURTHER WORK 105

a. Instructions As Fault Detectors 105
b. Fault Detection Circuit Considerations 105
c. Level of BIT Verification 107
d. Further Work 108

TABLE OF CONTENTS
(continued)

Page

7. REFERENCES 111

APPENDICIES 113

APPENDIX A 115

(1) PDP-11/70 ISP Description 115
(2) ALU Described in ISP at Gate Level 151
(3) ISP Description of the Memory Controller for the Dual-CPU Configuration 157
(4) Execution of Some of the PDP-11/70 Diagnostics 159

APPENDIX B - TI 74181 ALU 163

APPENDIX C 167

(1) The Two CFA Candidate Programs 167
(2) Functional Fault Injection in the Instruction Decode and Control Functional Module 173
(3) Functional Fault Injection in the Instruction Register 177
(4) Functional Fault Injection in the Data Manipulation Functional Module 179

LIST OF FIGURES

Figure No.	Title	Page
1	PDP-11/70: Functional Modules and Bus Structure	5
2	Jump-To Subroutine	7
3	Compare Operation	8
4	Index Addressing	10
5	ALU Description Written at the Instruction Set Level	16
6	Register File Model	20
7	Decode Schematic	21
8	Register File Description with Fault Injection	22
9	Dual CPU Model	25
10	Interconnection Description of the Dual CPU Used for Input to PMS	26
11	Read Entity for the Single-Processor Configuration	28
12	Read Entity for the Dual-Processor Configuration	29
13	Digital Systems Fault Propagation	33
14	The Relationship Between a Function and a Gate in the TI 74181 ALU	39
15	Classes of Faults in the TI 74181 ALU Simulation	41
16	Frequency of an ALU Function in Error When All the Gates Are Stuck-at-Zero or Stuck-at-One, One at a Time	48
17	Frequency of an ALU Function in Error When All the Gates Are Stuck-at-Zero, One at a Time	49

LIST OF FIGURES
(continued)

Figure No.	Title	Page
18	Frequency of an ALU Function in Error When All the Gates Are Stuck-at-One, One at a Time	50
19	Statistics Expressing the Relationship Between Functions and Gates in the 74181 ALU	53
20	Digital System Fault Propagation from Gate Level to the Functional Level	94
21	Digital Systems Fault Propagation from Gate Level to the Functional Level for Each Function	95
22	An OR Gate with Three Inputs	95
23	Breakdown of Number of Faults in the 74181 ALU	98
24	Functional Faults to Be Injected in the Presence of BIT	109
APPENDIX		
B1	TI 74181 ALU Logic Diagram	164

LIST OF TABLES

Table No.	Title	Page
1	TI 74181 ALU Functions	13
2	TI 74181 ALU Functions Used by PDP-11/70	14
3	Function Codes and Operations	15
4	Frequency and Percentage of Times a TI 74181 ALU Function Will Be in Error When a Gate Is Stuck-at-Zero	45
5	Frequency and Percentage of Times a TI 74181 ALU Function Will Be in Error When a Gate Is Stuck-at-One	46
6	Frequency and Percentage of Times a TI 74181 ALU Function Will Be in Error When a Gate is Either Stuck-at-Zero or One	47
7	Some Typical Instruction Execution Frequencies	51
8	Frequency of Error for Typical PDP-11 Instructions	51
9	CFA Programs	56
10	Wrong Register Is Selected	59
11	Register Is Not Selected	61
12	Desired Register Is Included	64
13	Desired Register Is Excluded	64
14	Possible Values of Variable FAULT and the Action Taken	77
15	Wrong Register Is Selected	84
16	Register Is Not Selected	84
17	Desired Register Is Included	85
18	Desired Register Is Excluded	85

LIST OF TABLES
(continued)

Table No.	Title	Page
19	Pin-Level Fault Manifestation for a Hypothetical ALU	96
20	Fault Manifestation for Dependent and Regular Faults	97
21	Results of Generating Parity for Different Combinations of Output Bits	106
22	Total Percentage Increase When Generating Parity for Two Sets of Bits	107
APPENDIX		
B1	TI 74181 ALU Functions	165

1. INTRODUCTION

a. Background

A previous study has addressed the identification of built-in-test (BIT) techniques for the Military Computer Family (MCF) [4]. In that study, appropriate BIT techniques were identified based upon an assumed fault population. A model was developed to predict where in the system these faults are most likely to occur. Based upon this model, a rationale was developed for deploying built-in-test resources and a unified BIT approach for MCF was recommended.

A later study focused on the identification of relevant BIT requirements for Military Computer Family form, fit and function (F^3) specifications [5]. This study identified elements of the MCF F^3 specifications that could be modified to accommodate the BIT features identified and recommended in the previous study. In addition to recommending the modification of MCF specifications, a method for validating BIT performance in vendor-supplied F^3 hardware was proposed. This is a particularly difficult problem since a basic premise of the MCF program is that the internal hardware and software modules procured under the F^3 specifications cannot be specified. The BIT performance specification and validation approach recommended was the use of Instruction Set Processor (ISP) language descriptions with a functional fault injection capability.

b. Present Study Objectives

The present study was concerned with extending the BIT performance assessment methodology proposed in reference [5] for verifying MCF built-in-test performance and the application of this methodology to a particular member of the Military Computer Family.

c. Study Approach

The recommended approach is to describe MCF functional modules using ISP language descriptions. The PDP-11/70 member was selected as representative. The PDP-11/70 modules were verified to insure their proper functional behavior and the previously recommended BIT approaches were described in ISP and applied to these modules.

In order to validate BIT performance for modules, a functional fault model was developed. The model developed in this study started with basic fault occurrences at the logic level and related these faults to their functional-level manifestations. This fault model was then exercised and the performance of the selected BIT approaches assessed.

d. Report Overview

This report presents the study results in five parts. Section 2 discusses hierarchical computer descriptions using ISP and points out some differences between this version and previously used ISP descriptions. Section 3 describes the development of a functional fault model for the PDP-11/70. This functional fault model was then applied to the ISP descriptions as described in Section 4. Section 5 discusses refinements made to the functional fault model. Section 6 summarizes the results of the study and recommends further work.

2. HIERARCHICAL COMPUTER DESCRIPTIONS USING ISP

a. Introduction

Instruction Set Processor (ISP) notation was first introduced by Bell and Newell [1] as a formalism to describe the programming level in the hierarchy of digital system descriptions. This notation was used mainly for publication purposes. A subset of ISP was later introduced by Barbacci [2] and used in design automation and architecture evaluation. ISPL was the tool used in the Computer Family Architecture (CFA) study [3]. ISPS, which is the current version of ISP, has been used to study built-in-tests (BIT) [4] and fault injection [5].

The ISP descriptions (for a single and a dual processor) discussed in this report are provided as accurate models of the PDP-11/70 computer architecture. They are used as test vehicles for research in digital systems and have been used to study built-in-tests (BIT) for the Military Computer Family (MCF) [4]. Currently, they are being used as a means to verify BIT and as a way to view the manifestations of functional faults at the system level.

b. History

ISP was first used in 1977 for computer architecture evaluation in the Computer Family Architecture (CFA) study [3]. From this study, several computer architectures were described in ISP -- among them the PDP-11/70. The PDP-11/70 description was in turn used by RTI as a model for studying BIT methodology [4]. In July 1979, Carnegie-Mellon University (CMU) released a new ISP description of the PDP-11/70 as an ongoing part of the CFA study. This description was an improvement over the previous one due to its clarity, modularity, structure and documentation. This CMU version is the basis for the single and dual PDP-11/70 processor descriptions that RTI is currently using and that are discussed in the following sections.

c. Differences Between the Research Triangle Institute and Carnegie-Mellon University ISP Descriptions of the PDP-11/70

Although the RTI model of the PDP-11/70 is based upon the CMU model, many differences exist. These differences stem from the point of view, or level of abstraction, at which the computer architecture is studied.

The CMU model was written to describe the architecture of the PDP-11/70 as seen by the machine-language programmer. CMU's definition of computer architecture, taken from Amdahl, Blaauw and Brooks [6], states that computer architecture is the attributes of a computer as seen by a machine-language programmer. The definition includes the instruction set, instruction format, operation codes, address modes, and all registers and memory locations which may be directly manipulated or tested by a machine-language program. This definition excludes the time or speed of any operation, bus structure, and electrical or physical organization.

RTI's model of the PDP-11/70 also describes the computer architecture as defined by Amdahl, Blaauw, and Brooks. However, the description is carried one step further; it includes the computer architecture based on a canonical bus structure. (See Figure 1.) Why the difference? CMU's model was written to evaluate a computer architecture, whereas RTI's model was written to study built-in-tests and functional faults, both of which require details concerning the bus structure. This does not mean that the CMU model does not have a bus structure -- it does. However, CMU's bus structure is not an explicitly stated part of the description, nor is it a conscious part of the description design. Instead, the bus structure is based, in an ad hoc manner, on the interconnection of the ISP entities. On the other hand, RTI's bus structure *is* an explicit part of the description and a very conscious part of the description design.

In order to understand some of the consequences of adding bus structure to the RTI computer architecture description, small segments of the ISP code from the CMU model and the RTI model are compared below. A knowledge of ISP for this comparison is helpful, but not necessary; a knowledge of some PDP-11 assembly language is more useful [7].

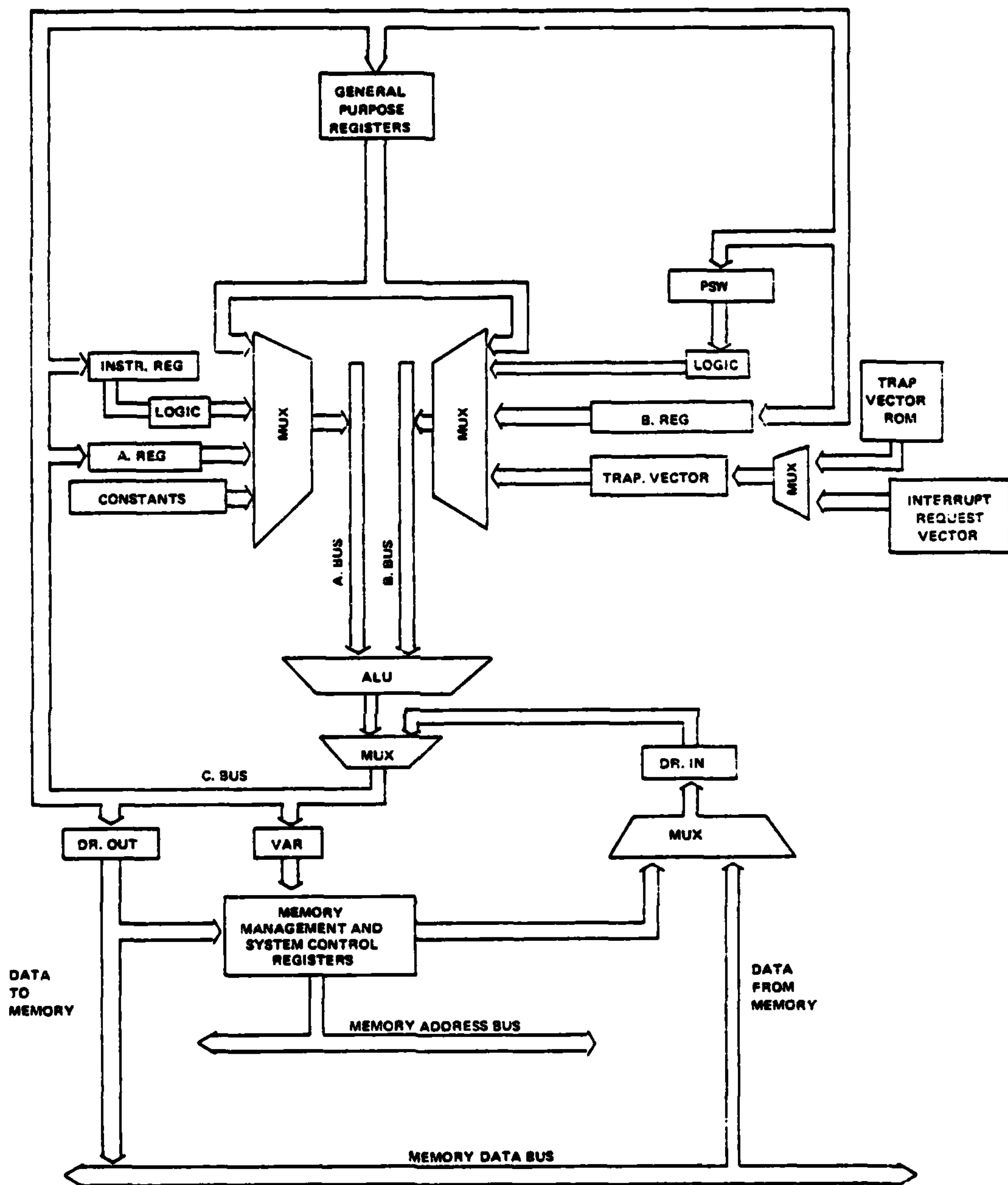


Figure 1. PDP-11/70: Functional Modules and Bus Structure.

Several ISP entities (identifiers and routines) will be encountered in some segments of the ISP code, some of which are common to both the CMU and the RTI models. The PDP-11 condition code bits, ISP identifiers, are N (negative result), V (overflow), Z (zero result) and C (carry). In the CMU model the registers, TEMP1 and TEMP2, are basically synonomous with the A.LATCH and B.LATCH in the RTI model. SRC.OPN(reg, flag) and DST.OPN(reg, flag) retrieve the source and destination operand, respectively, and put each into a register, which is generally either A.REG or B.REG. Both SRC.OPN and DST.OPN use the GET.OP (get operand) routine of the CMU model. R[index] is the register file. The two operations presented below are the Jump-to Subroutine (JSR reg,dst) and Compare (CMP src,dst). They are compared in the RTI and CMU models.

The first operation is a Jump-to subroutine called, JSR reg,dst. (See Figure 2.) The destination address is first evaluated (DST.ADDR) and stored in a temporary hardware register. The contents of "reg" are pushed onto the stack pointed at by the SP (stack pointer, register 6). "Reg" is then loaded with the current PC (program counter), the return link, and then the PC is set to the destination (subroutine) address being held in the temporary hardware register. Comparing the ISP codes between the two models, it is obvious that many more steps are required to perform this operation in the RTI model than in the CMU model. However, in order to use the extra level of detail (bus description) in the RTI model correctly and completely, these steps are a must.

Figure 3 presents the CMU and RTI versions of the second operation, the Compare operation. The CMU version retrieves the source and destination operands, putting them in TEMP1 and TEMP2, respectively. The destination operand is subtracted from the source operand, with the result moving to TEMP. The C and Z bits are set accordingly. A complicated maneuver is performed to set the V and N bits, depending on whether the operation is a byte or a word compare. In the RTI version, the source and destination operands are retrieved and stored, respectively, in A.REG and B.REG, and then latched to the A.LATCH and B.LATCH of the ALU. The ALU performs a subtraction. The condition code mechanism (CC.SET) then sets N, Z, V and C, depending on the result.

```
JSR: = ! Jump to subroutine, JSR opcode #004

     BEGIN
        TEMP = GET.OP.ADDRESS(DESMOD, DESREG, 2) next  ! Temp<--destination addr
        PUT.OP(#4, #6, Ø) = GET.OP(#0, SRCREG, Ø) next ! SP<--push(reg)
        PUT.OP(#0, SRCREG, Ø) = R[7] next              ! reg<--PC
        R[7] = TEMP                                    ! PC<--TEMP
        END
```

A) CMU Version of Jump-To Subroutine.

```
JSR: = ! Jump to subroutine, JSR opcode #004

     BEGIN
        DST.ADDR(PLACE.HOLDER) next B.REG<--PLACE.HOLDER next ! destination addr

        A.LATCH = A.BUS(R[GET.INDEX(CM,SRCREG)]) next ! Contents(Reg)
        ALU (TRNSF.A, Ø) next
        PUSH (ALU)       next ! SP <-- PUSH(reg)

        B.LATCH <-- B.BUS(PC) next
        ALU(TRNSF.B, Ø)       next
        R[GET.INDEX(CM,SRCREG)] = C.BUS(ALU) next ! REG <-- PC

        B.LATCH = B.BUS(B.REG) next ! destination address
        ALU (TRNSF.B, Ø)        next
        PC = C.BUS(ALU)       ! PC <-- destination address

     END
```

B) RTI Version of Jump-To Subroutine.

Figure 2. Jump-To Subroutine.

```
CMP: = ! Compare and Compare Byte
       ! CMP opcode #02, CMPB opcode #12

     BEGIN
        TEMP1 = GET.OP(SRCMOD, SRCREG, BYOP) next ! source opn
        TEMP2 = GET.OP(DESMOD, DESREG, BYOP) next ! destination opn

     C@TEMP = TEMP1-TEMP2 next ! source-destination

! set condition codes

     Z = TEMP EQL Ø next ! result = Ø
     DECODE BYOP
        BEGIN
           BEGIN
           V = (TEMP<15> EQL TEMP2<15>) and (TEMP1<15> XOR TEMP2<15>);
           N = TEMP<15>
           END
           BEGIN
           V = TEMP<8> XOR TEMP<7>;
           N = TEMP<7>
           END
        END
     END,
```

A) CMU Version of Compare Operation.

```
CMP: = !Compare and Compare Byte
       !CMP opcode #02, CMP opcode #12


     BEGIN
        SRC.OPN(A.REG, BYOP) next ! source opn
        DST.OPN(B.REG, BYOP) next ! destination opn
        A.LATCH <-- A.BUS(A.REG) ; B.LATCH <-- B.BUS(B.REG) next

           ALU(SUB., BYOP) next ! source-destination

        CC.SET (NZVC, BYOP, CMP.) ! set condition codes N, Z, V and C
     END,
```

B) RTI Version of Compare Operation.

Figure 3. Compare Operation.

Considering the ISP code presented in Figure 3, it first appears that the RTI Compare operation is shorter than the CMU version. It is not. To understand the difference, it is necessary to examine the Compare operation more closely. DST.OPN and SRC.OPN both use GET.OP (retrieves the operand) as set up in the CMU code. GET.OP calls on GET.OP.ADDRESS, which calculates the operand address.* GET.OP. ADDRESS uses the operand address mode to index into a segment of code that performs the address calculation for that address mode.

The code required to do index addressing (address mode #6) is presented below. In the instruction CLR 5(R2) the address to be cleared (EA) is calculated as

EA = 5 + (contents of register 2).

The code in memory appears like this:

Relative Address	Instruction	
ØØØØ2	ØØ5Ø62	; CLR
ØØØØ4	ØØØØØ5	; index

The index appears in the word immediately following the CLR instruction. To see how this is done in both versions of the ISP code, see Figure 4. As can be seen from the ISP code, there is a great deal more work involved in the RTI version than in the CMU version to retrieve operand address. This additional level of work is reflected throughout the RTI description.

Another difference in the Compare operation in the two versions occurs in setting/clearing the condition codes. Within each segment of ISP code for an instruction, the CMU model evaluates and then sets/clears the condition code bits. The RTI version, on the other hand, uses a collection of logic common to all the operations to set/clear the condition codes based on the operation code (opcode). This is more realistic than the CMU version.

*There are eight address modes in the PDP-11. See reference 6 for additional information.

```
6: = BEGIN                         !index addressing
     GET.OP.ADDRESS = '1@ R[7] next     !address of index  in I space
     R[7] = R[7]+2 ;                   !point next instruction
     SETMM1(2,7)   next
     GET.OP.ADDRESS = (READ(CM,Ø,GET.OP.ADDRESS) + R[Get.Index]<15:0>
     END,
```

A) CMU Version for Calculating an Indexed Address.

```
6: = BEGIN                         !index addressing
     A.LATCH = A.BUS(CONST2); B.LATCH = B.BUS(PC)  next
     VAR = I.SPACE@C.BUS(ALU(TRNSF.B,Ø))          next !Address of index
     PC = C.BUS(ALU(ADD.,Ø))  next ! update PC
     SETMM1(2,7)  next
     DR.IN = READ(CM,Ø)  next ! get index
     ALU.REG = C.BUS(DR.IN)  next
     A.LATCH = A.BUS(ALU.REG) ; B.LATCH = B.BUS(R[Get.Index])  next
     Result.Req = D.SPACE@C.BUS(ALU(ADD.,Ø))
     END,
```

B) RTI Version for Calculating an Indexed Address.

Figure 4. Index Addressing.

By writing the ISP description in terms of functional units (i.e., A-BUS, ALU), parts of the description can be quickly modified without affecting the descriptions of other functional units. One reason for this approach is that load module size increases and simulation speed decreases as more detail is added to the description. It is, therefore, desirable to have the level of detail where it is needed and to use another higher level of abstraction (less detail) where detail is of no importance. If the description is written in terms of functional modules, the modules can then be interchanged, depending upon the detail required for the experiment.

An example of this application is the memory management unit (MMU). In the block diagram of Figure 1 the MMU is a functional module. MMU has not played any role in the areas RTI is currently investigating; thus, a detailed ISP description of the MMU at this point would add an unnecessary increase in load module size and decrease in simulation speed. For this reason, the same code for the MMU in the CMU model is used in the RTI model. If more detail is needed, another description can be written, with the level of detail that is required, to replace the current, less detailed description.

Another example of this application is the ALU. Both versions are composed of several routines; the arithmetic/logic functions, performed by a TI 74181 ALU chip; the shift/rotate/byte swap functions, performed by extra logic; and the ISP code, used for injecting faults in the ALU module. RTI has two versions of the ALU module, one to describe the operations of the TI 74181 chip at the functional level; that is, in terms of arithmetic and logic operations, and the other, to describe the TI 74181 chip at the gate level [8]. The ISP code for the shift/rotate/byte swap functions remains the same in both ALU versions, but the fault injection code is different for each version. (Both versions are interchangeable in the PDP-11/70 description.) The functional-level description is fast and small in size but does not allow very detailed fault injection. The gate-level description is considerably slower. Simulation speed increases by a factor of 10 and is much larger in size, but any gate in the description can be faulted.

d. Single and Dual Processor Models

Since the single processor model is the basis for the dual processor model in RTI's ISP description, the discussion here will concentrate on the single processor model. A short discussion concerning the changes for the dual processor model is included.

Returning to Figure 1 and reviewing the ISP description in Appendix A(1), it is obvious that the ALU is involved in almost every data transfer or data manipulation activity. The ALU module is based on the TI 74181 chip, with a few non-TI 74181 functions included for shifting and byte swapping operations. The TI 74181 chip was chosen for two reasons. First, it is used in most PDP-11s. As a result and by studying DEC's engineering drawings [9], it was possible to find the subset of the TI's ALU functions used in the PDP-11/70. Second, the TI 74181 is one of the few ALU chips for which the logic diagram of the chip is available. Using this logic diagram, RTI wrote a simulator of the ALU chip which was used for extensive fault investigations. The results from these investigations were used to study functional faults and built-in tests in both the single and dual processor models.

Table 1 lists the entire set of operations that the TI 74181 ALU can perform, while Table 2 lists the subset of these operations used by the PDP-11/70. Table 3 presents the binary and octal function codes and the functions used in the RTI ALU modules. Note that the starred (*) functions are not used in the RTI CPU and are only executed if either line M or $\overline{C}_n$ is stuck-at-zero or stuck-at-one. For instance, if the ALU function code is 45_8 (A plus B) and line $\overline{C}_n$ is stuck-at-zero, then function A plus B plus one is executed. In the functional ALU description, not all function codes (2^6 or 64) are defined.

The ISP for the functional-level ALU is given in Figure 5. The ALU has four distinct segments of code. The first segment of code describes the TI functions to be used by the CPU (the non-starred functions). The second segment of code consists of functions that will not be executed unless line M or $\overline{C}_n$ is stuck-at-zero or stuck-at-one (the starred functions). The third segment of code is used to simulate the manifestations of certain gate-level faults at the output of the TI 74181 chip. The fourth segment of code describes the shift/rotate and byte swap functions.

Table 1. TI 74181 ALU Functions.

Selection				ACTIVE - HIGH DATA		
					M = L; Arithmetic Operations	
S3	S2	S1	S0	M = H Logic Functions	$\overline{C}_n$ = H (no carry)	$\overline{C}_n$ = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A plus 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) plus 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ plus 1
L	L	H	H	F = 0	F = minus 1 (2's compl)	F = Zero
L	H	L	L	$F = \overline{AB}$	F = A plus $A\overline{B}$	F = A plus $A\overline{B}$ plus 1
L	H	L	H	$F = \overline{B}$	F = (A + B) plus $A\overline{B}$	F = (A + B) plus $A\overline{B}$ plus 1
L	H	H	L	$F = A \oplus B$	F = A minus B minus 1	F = A minus B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ minus 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A plus AB	F = A plus AB plus 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A plus B	F = A plus B plus 1
H	L	H	L	F = B	$F = (A + \overline{B})$ plus AB	$F = (A + \overline{B})$ plus AB plus 1
H	L	H	H	F = AB	F = AB minus 1	F = AB
H	H	L	L	F = 1	F = A plus A*	F = A plus A plus 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) plus A	F = (A + B) plus A plus 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ plus A	$F = (A + \overline{B})$ plus A plus 1
H	H	H	H	F = A	F = A minus 1	F = A

Table 2. TI 74181 ALU Functions Used by PDP-11/70.

Selection					M = L; Arithmetic Operations	
S3	S2	S1	SØ	M = H Logic Functions	$\overline{C}_n$ = H	$\overline{C}_n$ = L
L	L	L	L	$F = \overline{A}$	F = A	F = A plus 1
L	L	H	H	F = Ø	F = minus 1 (2's comp)	F = Zero
L	H	H	L	$F = A \oplus B$	F = A minus B minus 1	F = A minus B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ minus 1	$F = A\overline{B}$
H	L	L	H	$F = \overline{A \oplus B}$	F = A plus B	F = A plus B plus 1
H	L	H	L	F = B	$F = (A + \overline{B})$ plus AB	$F = (A + \overline{B})$ plus AB plus 1
H	L	H	H	F = AB	F = AB minus 1	F = AB
H	H	L	L	F = 1	F = A plus A	F = A plus A plus 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ plus A	$F = (A + \overline{B})$ plus A plus 1
H	H	H	H	F = A	F = A minus 1	F = A

Table 3. Function Codes and Operations.

S_3	S_2	S_1	$S_Ø$	M	$\overline{C}_n$		Function F:
0	0	0	0	0	0	#0	F = A+1
0	0	0	0	0	1	#1	F = A
0	0	0	0	1	X	#02,03	F = $\overline{A}$
0	0	1	1	0	0	#14	F = Ø (zero)
0	0	1	1	0	1	#15	*F = minus 1
0	0	1	1	1	X	#16,17	F = Ø (zero)
0	1	1	0	0	0	#30	F = A minus B
0	1	1	0	0	1	#31	*F = A minus B minus 1
0	1	1	0	1	X	#32,33	F = A XOR B
0	1	1	1	0	0	#34	F = A and not B
0	1	1	1	0	1	#35	*F = A and not B minus 1
0	1	1	1	1	X	#36,37	F = A and not B
1	0	0	1	0	0	#44	*F = A plus B plus one
1	0	0	1	0	1	#45	F = A plus B
1	0	0	1	1	X	#46,47	*F = Not (A XOR B)
1	0	1	0	0	0	#50	*F = (A or not B) plus (A and B) plus 1
1	0	1	0	0	1	#51	*F = (A or not B) plus (A and B)
1	0	1	0	1	X	#52,53	F = B
1	0	1	0	0	0	#54	F = A and B
1	0	1	0	0	1	#55	*F = (A and B) minus 1
1	0	1	0	1	X	#56,57	F = A and B
1	1	1	0	0	0	#70	*F = (A or not B) plus 1
1	1	1	0	0	1	#71	*F = (A or not B) plus A
1	1	1	0	1	X	#72,73	F = A or B
1	1	1	1	0	0	#74	F = A
1	1	1	1	0	1	#75	F = A minus 1
1	1	1	1	1	X	#76,77	F = A

* Functions that are executed only if the control lines M and $\overline{C}_n$ are faulted.

X = Do not care.

```
ALU(FUNCT<5:0>,BYOP<>)<16:0>:= ! BIT 16 IS FOR THE CARRY BIT
BEGIN
**FAULT.DEFN**
FAULT<1:0>, ! SPECIAL FAULT TYPE
MASK<15:0>,  ! CONSTANT MASK

MAIN ALU.ENTRY:=
BEGIN (TC)
   ALU=ZEROS<16:0> NEXT ! CLEAN UP DETAIL
DECODE FUNCT =>
BEGIN

! THIS SECTION OF CODE DESCRIBES THE SUBSET OF THE TI 74181 ALU CHIP
! USED IN THE PDP11/70
OTHERWISE:= BEGIN
BEGIN

  [#1,#74,#76,#77]:= ALU=A.LATCH,
  [#52,#53]:= ALU=B.LATCH,
  [#14,#16,#17]  := ALU=#0,


  #30    := DECODE  BYOP =>
            BEGIN
            END,
            '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>
                                        +(NOT B.LATCH<BYTE.SIZE>)+'01
            END,
  #45    := DECODE BYOP =>
            BEGIN
            '0:=  ALU=A.LATCH+B.LATCH,
            '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>+B.LATCH<BYTE.SIZE>
            END,
   #00 := DECODE BYOP =>
          BEGIN
          '0:=  ALU=A.LATCH + #1,
          '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>+ #1<BYTE.SIZE>
         END,
   #75 := DECODE BYOP =>
          BEGIN
          '0:=  ALU=A.LATCH-#01,
          '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>-#1<BYTE.SIZE>
          END,
```

Figure 5. ALU Description Written at the Instruction Set Level. (continued)

```
  [#54,#56,#57]  := ALU=A.LATCH AND B.LATCH,
  [#72,#73]   := ALU=A.LATCH OR B.LATCH,
  [#34,#36,#37]:=ALU=  A.LATCH AND  NOT B.LATCH,
  [#02,#03]  :=ALU=NOT A.LATCH,
  [#32,#33]  := ALU=A.LATCH XOR B.LATCH, ! EXCLUSIVE OR

! THESE FOLLOWING INSTRUCTIONS SHOULD NEVER BE USED BY THE CPU BUT
! IF THERE IS AN ERROR IN THE FUNCTION SOME OF THESE CAN BE EXECUTED

 #15  := ALU<='111, ! MINUS 1
 #31  := ALU= A.LATCH - B.LATCH -1,
 #35  := ALU= (A.LATCH AND NOT B.LATCH) - 1,
 #44  := ALU= (A.LATCH + B.LATCH) + 1,
 [#46:#47] := ALU= NOT ( A.LATCH XOR B.LATCH),
 #55  := ALU= ( A.LATCH AND B.LATCH) - 1,
 #71  := ALU= (A.LATCH OR NOT B.LATCH) + A.LATCH,
 #70  := ALU= (A.LATCH OR NOT B.LATCH) + 1,
 #51  := ALU= (A.LATCH OR NOT B.LATCH) + (A.LATCH AND B.LATCH),
 #50  := ALU= (A.LATCH OR NOT B.LATCH) + (A.LATCH AND B.LATCH) + 1,

OTHERWISE:=STOP()    ! TROUBLE UNDEFINED FUNCTION CODES

END NEXT
! THIS CODE IS USED TO INJECT FUNCTIONAL FAULTS IN THE ALU OUTPUT
! THESE SPECIFIC FAULTS WERE DERIVED FROM ANALYSIS DONE FROM SIMULATION
! DATA OF THE TI 74181 CHIP
   DECODE FAULT =>
   BEGIN
0:= NO.OP(),
1:= ALU<15:0>=ALU<15:0> XOR MASK,
2:= ALU<15:0>=ALU<15:0> + MASK,
3:= ALU<15:0>=ALU<15:0>- MASK
   END

END,
```

Figure 5. ALU Description Written at the Instruction Set Level. (continued)

```
! THIS CODE DESCRIBES THE SHIFT/ROTATE/SWAP MECHANISM IN THE ALU. THIS
! IS NOT PART OF THE TI 74181 CHIP
[SWAB.,ROR.,ROL.,ASR.,ASL.]:=
DECODE FUNCT =>
BEGIN
  SWAB. := ALU=A.LATCH<7:0>@A.LATCH<15:8>,
  ROR.  := DECODE BYOP =>
           BEGIN
           '0:= ALU=(B.LATCH<0>@A.LATCH) SRR 1,
           '1:= ALU<16>@ALU<7:0>=(B.LATCH<0>@A.LATCH<7:0>) SRR 1
           END,
  ROL.  := DECODE BYOP  =>
           BEGIN
           '0:= ALU=A.LATCH@B.LATCH<0>,
           '1:= ALU<16>@ALU<7:0>=A.LATCH<7:0>@B.LATCH<0>
           END,
  ASR.  := DECODE BYOP =>
           BEGIN
           '0:=(ALU<15:0><=A.LATCH<15:1> ; ALU<16>=A.LATCH<0>),
           '1:=(ALU<7:0> <=A.LATCH<7:1>  ; ALU<16> =A.LATCH<0>)
           END,
  ASL. :=  DECODE BYOP =>
           BEGIN
           '0:= ALU=A.LATCH@'0,
           '1:=( ALU<7:0>= A.LATCH<6:0>@'0; ALU<16>=A.LATCH<7>)
           END


END
END
END,
```

Figure 5. ALU Description Written at Instruction Set Level.

The segment of code used for injecting faults into the ALU module is derived from extensive simulations of the TI 74181 chip. (See Section 3b(5) for more detail). As a result of these simulations, several fault manifestations occurred. One of the fault manifestations consists of either exclusive ORing a constant or adding/subtracting a constant to the output of the TI chip. ISP code in the ALU module was written to do this. These operations and constants are controlled by two global parameters in the ALU module called MASK and FAULT.

As an adjunct to RTI's studies of functional faults, a gate-level ISP description of the TI 74181 chip was written (See Appendix A(2)) to observe the manifestation of gate-level faults at the system level. Note that this added description decreases simulation speed. It takes approximately 10 times as long to run a simulation using the gate-level ALU as it takes to run a simulation of the functional-level ALU.

Another important functional unit in the PDP-11/70 is the register file R. In the CMU version and earlier RTI versions [4], this register file was simply an array indexed by a register number. This has been modified rather drastically for the purpose of studying functional faults. The register file R in the current RTI version is based on a model (Figure 6) defined by Abraham [10] and expanded by Joobbani in Section 3b(1) of this report. The register decode functional faults simulated were: 1) no register is selected, 2) the wrong register is selected, or 3) multiple registers are selected. These three types of faults can be modeled by failures in the decoder.

Figure 7 is a functional block diagram of an SN54154 4-to-16-line decoder [8]. For the no-register-selected case, if the inputs are L, L, L, H, then the active line, high (H), should be one. If a stuck-at-high exists on the output of the NAND gate, none of the lines are active and the result of reading the register file is whatever is in the buffer at the time (wrong-register-selected fault). For the multiple-register-selected case, several lines could be active at once. Consequently, several registers would be read/written simultaneously. This register file model is simulated by the ISP code given in Figure 8. The code has some hooks (FAULT.SETTING), which allow all these functional faults to be simulated at run time.

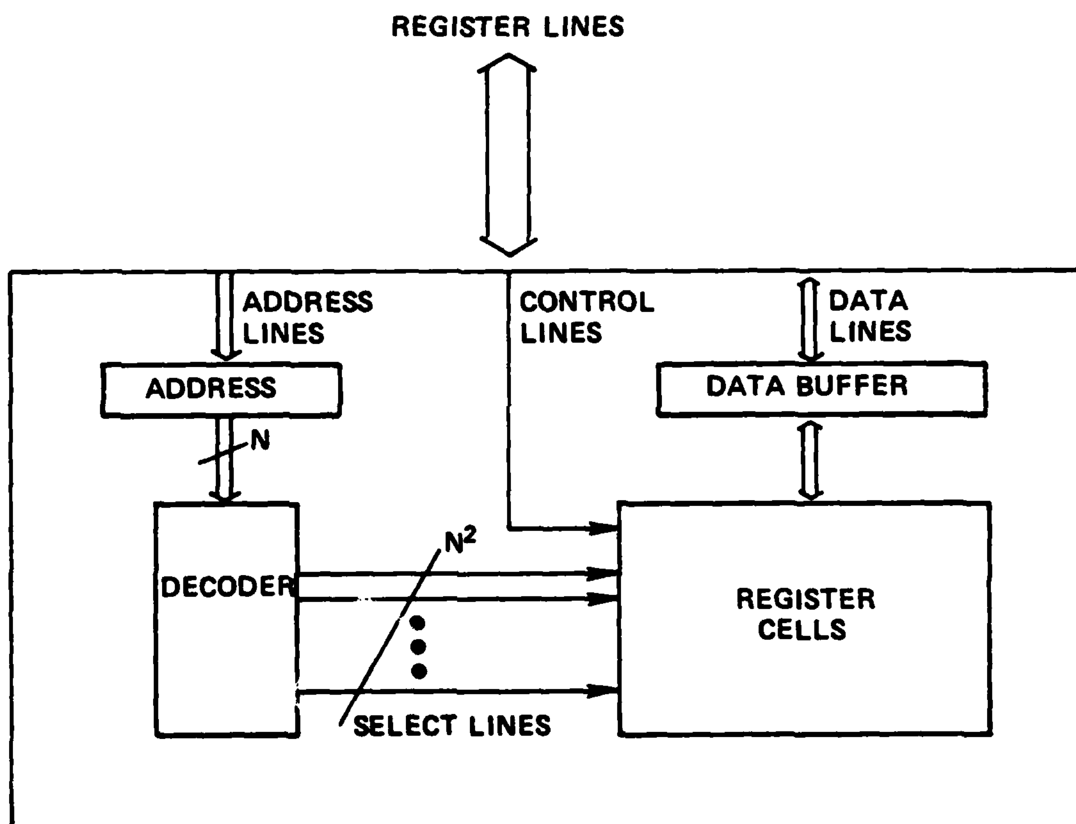


Figure 6. Register File Model.

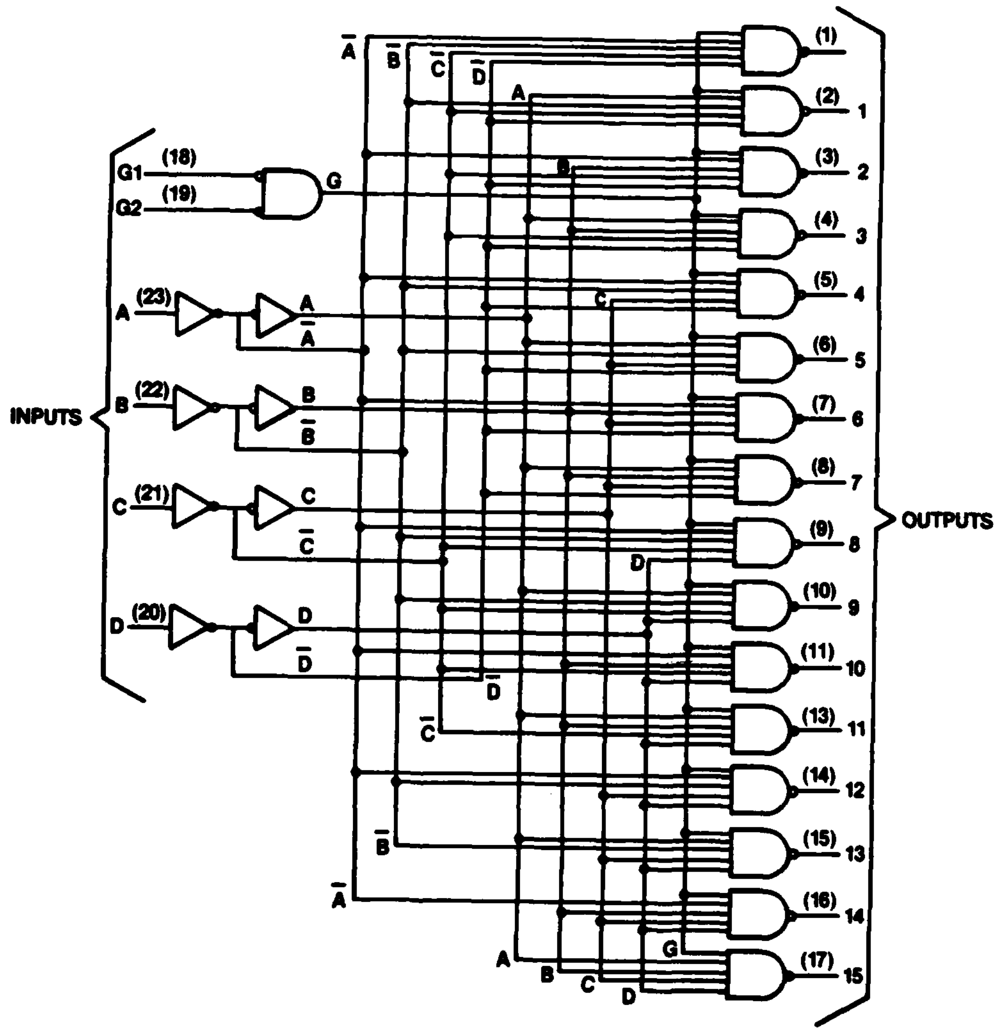


Figure 7. Decode Schematic.

```
R(INDX<3:0>,R.W<>)<15:0>:=
BEGIN
**LOCAL.DEFN**
REG[15:0]<15:0>,

FAULT.SETTING<11:0>,
  INCL<>       :=FAULT.SETTING<11>, ! INCL
  MULT.SEL<>  :=FAULT.SETTING<10>,
  NO.SEL<>    :=FAULT.SETTING<9>,
  AND.OR<>   :=FAULT.SETTING<8>, ! AND=1, OR=0
  R.S/SELECTED.REG<7:0> :=FAULT.SETTING<7:0>,
MACRO WRT:=| '0| ,
MACRO RD:=| '1| ,

MAIN REG.ENTRY:=
BEGIN

  DECODE R.W =>
  BEGIN
WRT:=  DECODE FAULT.SETTING<10:9> =>
        BEGIN
'00:=   REG[INDX]=R,
'01:=   IF (R.S SRO INDX)<0> EQLU '0 => REG[INDX]=R,
'10:=   BEGIN
          IF R.S<0> => REG[0]=R;
          IF R.S<1> => REG[1]=R;
          IF R.S<2> => REG[2]=R;
          IF R.S<3> => REG[3]=R;
          IF R.S<4> => REG[4]=R;
          IF R.S<5> => REG[5]=R;
          IF R.S<6> => REG[6]=R;
          IF R.S<7> => REG[7]=R NEXT
          IF INCL => REG[INDX]=R
        END,
'11:=   STOP()
       END,

RD:=   DECODE FAULT.SETTING<10:9> =>
        BEGIN
'00:=   R=REG[INDX],
'01:=   IF (R.S SRO INDX)<0> EQLU '0 => R=REG[INDX],
```

Figure 8. Register File Description with Fault Injection.
(continued)

```
'10:=    DECODE AND.OR =>
           BEGIN
'0/OR:= BEGIN
           R<='0 NEXT
           IF R.S<0> => R=R OR REG[0];
           IF R.S<1> => R=R OR REG[1];
           IF R.S<2> => R=R OR REG[2];
           IF R.S<3> => R=R OR REG[3];
           IF R.S<4> => R=R OR REG[4];
           IF R.S<5> => R=R OR REG[5];

           IF R.S<6> => R=R OR REG[6];

           IF R.S<7> => R=R OR REG[7];
           IF INCL   => R=R OR REG[INDX]
        END,
'1/AND:=BEGIN
           R<='1 NEXT
           IF R.S<0> => R=R AND REG[0];
           IF R.S<1> => R=R AND REG[1];
           IF R.S<2> => R=R AND REG[2];
           IF R.S<3> => R=R AND REG[3];
           IF R.S<4> => R=R AND REG[4];
           IF R.S<5> => R=R AND REG[5];
           IF R.S<6> => R=R AND REG[6];
           IF R.S<7> => R=R AND REG[7];
           IF INCL   => R=R AND REG[INDX]
        END


           END,
'11:=   STOP()
        END
  END
END
END,
```

Figure 8. Register File Description with Fault Injection.

The registers attached to the busses are just that, registers. The only special ones are the Program Status register (PS) and the Instruction register (I), which have some combinational logic used to place certain subfields of these registers onto their respective busses. An example of this is the macro IR.CC, which mimics the logic by placing the low-order four bits of the Instruction register (I) onto BUS.A. These bits determine which condition code bits in the Program Status register (PS) are set or cleared. The busses, A.BUS, B.BUS, and C.BUS, are all currently implemented as macros. If modification of these busses to include BIT is desired, it can be done by replacing the macros with routines containing the desired characteristics. Using macros now provides for a smaller load module and a faster simulation time.

The dual-CPU model was built using a hierarchical structuring program called PMS [11]. The PMS program is based upon the PMS notation introduced by Bell and Newell [1] for describing the interconnection of processors (P), memories (M), and switches (S). The dual-CPU configuration of Figure 9 uses two CPUs and shares a common memory. The major differences between the single-CPU and dual-CPU descriptions are:

1. Each processor in the dual-CPU has been augmented by an explicitly defined memory bus. A synchronizing flag called the bus ready flag has also been defined.

2. Memory has been removed from the dual-CPU description and has been made a distinct module. The memory module has also been given some intelligence called the memory controller.

Three modules are connected together. Two of these are CPU modules with each unit possessing its own processor, memory management unit, and I/O memory. The third module is the memory module. The input to the PMS program, which describes the interconnection of the modules, is seen in Figure 10.

The following is a description of how the dual CPU works. The synchronizing flag is set by one CPU, after it has put information on the memory bus (i.e., address, data, control), to inform the memory controller that it wishes to make a memory request. The CPU then goes into a WAIT state until the synchronizing flag is cleared by the memory controller. The CPU checks the error line on its bus to see if an error has occurred

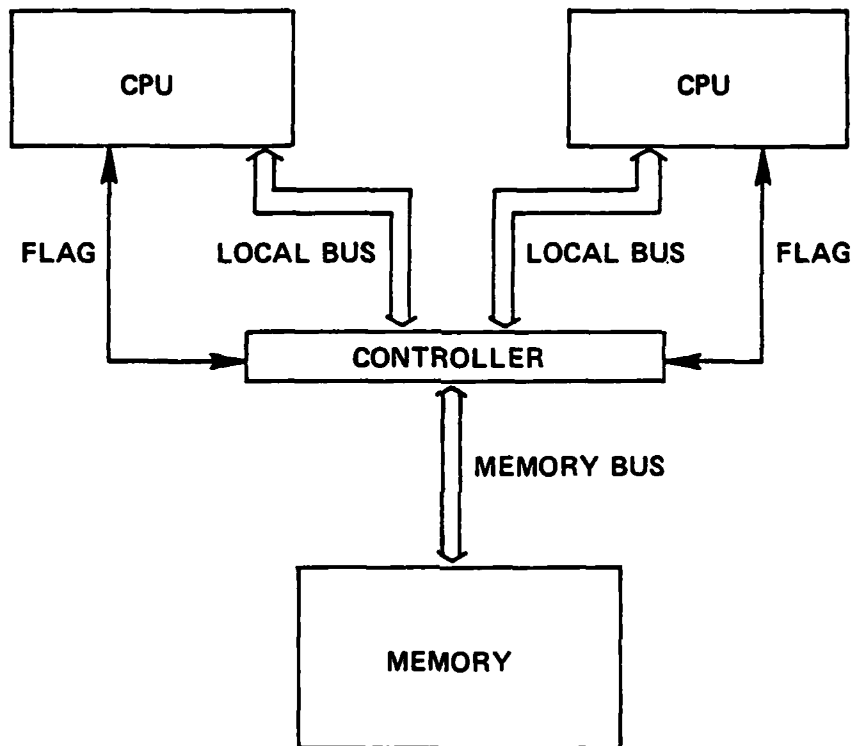


Figure 9. Dual CPU Model.

```
MODULE PDP11:=
     CPUA:=TEMP:PDP2.GDB
     CPUB:=TEMP:PDP2.GDB
     MEMORY:=TEMP:MEMORY.GDB

       CONNECT CPUA(BUS.RDY.FLAG<>):=MEMORY(BUS.A.RDY<>)
       CONNECT CPUA(BUS<41:0>)     :=MEMORY(BUS.A<41:0>)
       CONNECT CPUB(BUS.RDY.FLAG<>):=MEMORY(BUS.B.RDY<>)
       CONNECT CPUB(BUS<41:0>)     :=MEMORY(BUS.B<41:0>)
```

Figure 10. Interconnection Description of the Dual CPU Used for Input to PMS.

while accessing the memory. If an error has occurred, the CPU begins an error-handling routine to decide what to do next. If no error has occurred, and if a read operation has been requested, then the CPU takes the data off the bus and continues. Figures 11 and 12 show part of the ISP code for requesting a memory read for both the single and dual processor.

The memory controller waits until both synchronizing flags are set by their respective CPUs. Parity is checked for both busses for agreement, bit by bit. If the parity is wrong, the request is suppressed and a parity error is returned. If the busses are in agreement, a copy of the bus is sent to memory to perform the requested operation. After the memory operation is performed, the busses are updated with data from a read operation and/or error information operation. If a read operation occurs, the synchronizing flags are cleared and control is returned to the CPUs. If the busses are in disagreement, the memory operation is suppressed, and an error line on both busses is set. The synchronizing flags are then cleared and control returns to the CPUs. Appendix A(3) has the ISP code for the memory controller.

Two types of memory-related errors can occur. The first, the parity error discussed above, can occur in transmission or in reading or writing the memory. This can be handled by the CPUs in several standard ways. The second type is called an out-of-sequence error (i.e., accesses to different memory locations are requested), and it is much more serious than the first type. If the CPUs are to proceed, then a common sequence must be asserted for each CPU; for example, by using a rollback routine.

Currently, RTI is not investigating methods for handling out-of-sequence errors, but is using the dual CPU as a BIT technique for detecting and isolating faults. The data RTI generated from running programs on the dual-CPU configuration will hopefully provide some useful information on fault latency, and will produce information allowing for the comparison of system-level fault manifestations as they occur in the single and dual CPU.

Both the single- and dual-CPU configurations were tested to demonstrate that they had a PDP-11/70 architecture. This was done by running diagnostic programs generated by the CFA project for verifying computer descriptions. Appendix A(4) presents some of the results.

```
READ(MODE<1:0>,BYTE.ACCESS<>)<WORD.SIZE> :=
        BEGIN
        IF VAR<0> AND NOT BYTE.ACCESS => ODDERR() NEXT
        VIRT.PHY(MODE,VAR,0) NEXT
        DECODE VIRT.PHY<21:18> EQLU #17 =>
                BEGIN
                0 := DECODE BYTE.ACCESS =>
                     BEGIN
                     READ=MW[VIRT.PHY],
                     READ<=MB[VIRT.PHY]   ! SIGN EXTEND
                     END,
                1 := DECODE VIRT.PHY<17:0> =>
                        BEGIN
                     PSW.:= DECODE BYTE.ACCESS=>
                             BEGIN
                             READ=PS,
                             READ<=PS<BYTE.SIZE>
                             END,
                   GPREG  := DECODE BYTE.ACCESS =>
                             BEGIN
                             READ=R.OUT(VIRT.PHY<3:0>),
                             READ<=R.OUT(VIRT.PHY<3:0>)<BYTE.SIZE>
                             END,
                      OTHERWISE:= DECODE BYTE.ACCESS =>
                                   BEGIN
                                   READ = MWIO[VIRT.PHY],
                                   READ<= MBIO[VIRT.PHY]
                                  END
                        END
                END
          END,
```

Figure 11. Read Entity for the Single-Processor Configuration.

```
READ(MODE<1:0>,BYTE.ACCESS<>)<WORD.SIZE> :=
        BEGIN
        IF VAR<0> AND NOT BYTE.ACCESS => ODDERR() NEXT
        VIRT.PHY(MODE,VAR,0) NEXT
        DECODE VIRT.PHY<21:18> EQLU #17 =>
                BEGIN
                0 := BEGIN
                     MEMORY.BUS='00@BYTE.ACCESS@ZEROS<15:0>@VIRT.PHY@RD NEXT
                     BUS.RDY.FLAG=TRUE NEXT

                        WAIT( NOT BUS.RDY.FLAG ) NEXT

                     IF MEMORY.BUS<MISMATCH> => ERROR.HANDLER NEXT
                     READ_MEMORY.BUS<DATA.LINES>
                     END,
                1 := DECODE VIRT.PHY<17:0> =>
                        BEGIN
                     PSW.:= DECODE BYTE.ACCESS=>
                             BEGIN
                             READ=PS,
                             READ<=PS<BYTE.SIZE>
                             END,
                   GPREG  := DECODE BYTE.ACCESS =>
                             BEGIN
                             READ=R.OUT(VIRT.PHY<3:0>),
                             READ<=R.OUT(VIRT.PHY<3:0>)<BYTE.SIZE>
                             END,
                      OTHERWISE:= DECODE BYTE.ACCESS =>
                                         BEGIN
                                         READ = MWIO[VIRT.PHY],
                                         READ<= MBIO[VIRT.PHY]
                                         END
                        END
                END
         END,
```

Figure 12. Read Entity for the Dual-Processor Configuration.

3. FUNCTIONAL FAULT MODELING

a. Introduction

With the advent of high-density integrated circuit technology, multiple implementations of computer architectures have become available. The use of gate-level logic to generate test programs for such architectures is not practical because of the detailed information required to describe each implementation. Recently, efforts have focused on functional testing [10, 13,14]. An obvious advantage of functional testing is that test programs which are generated for a given architecture can be used for testing each implementation of that architecture, free from the detailed information about each implementation.

The work described here addresses functional fault modeling. In this discussion, functional fault modeling refers to the classification of gate-level faults into functional faults. In other words, functional faults are the manifestation of gate-level faults at a functional level.

To understand fault manifestations at a functional level, it is necessary to start with the basic elements of a digital system. These basic elements include the resistor and the transistor, henceforth referred to as components. The occurrence of a fault at the component level; e.g., a resistor which is burned out or shorted, can result in a stuck-at-zero or stuck-at-one condition at the gate level. This stuck-at condition is the manifestation of a fault at the component level. On a higher level; e.g., the chip level, the occurrence of a fault at the gate level can manifest itself as a chip pin being stuck-at-zero or stuck-at-one, or two pins shorted together. With these types of occurrences in mind, it is possible to understand how a gate-level fault can affect the functionality of a system.

For example, a faulted gate used in a memory chip may cause one of the memory bits in that chip to be in error. If there is a parity checker in the system, the gate-level fault (stuck-at-zero or stuck-at-one) manifests itself as a parity error at the system level. If there is no parity checker and the word of the memory that has the faulted gate contains an instruction, a nonexistent instruction or an address error occurs. This, again, is a gate-level fault manifested at the system or functional level.

Figure 13 illustrates the concept of fault propagation in a digital system. As shown, gate-level faults map into a higher level, the chip level. Chip-level faults are represented by α_j and gate-level faults are represented by β_i, such that $\alpha_j = f(\beta_{i1}, \beta_{i2}, \beta_{im})$. Theoretically, the C-space is much smaller than the G-space. In the same manner, the C-space maps into the F-space (functional faults) by $\gamma_k = g(\alpha_{k1}, \alpha_{k2}, \ldots, \alpha_{km})$. Again, the F-space is theoretically smaller than the C-space. The F-space may be partitioned into functional modules. For example, if the F-space is a CPU, then it can be partitioned into register decode, instruction decode, data storage, data transfer, and data manipulation functions. Whether a fault is classified as a functional fault depends on what is defined as a function. For example, if an ALU adder is considered, a chip fault will be a functional fault. The mapping of gate-level faults into functional faults can potentially reduce the number of fault classes considerably.

This report is primarily concerned with functional fault modeling for maintenance purposes. Initial testing of a computer architecture was not considered, since it was assumed that the implementation had already been successfully tested. RTI's objective is to understand better the functional fault model, to see what relationships exist between physical faults and functional faults. The model defined here is derived using the PDP-11 family. Except for the portion of the model used for instruction decode and control functions, the model can be used with other computer architectures.

b. Functional Fault Modeling for the PDP-11 Family

The work discussed here began with the methodology for microprocessor functional testing proposed by Abraham [10,13]. Abraham's methodology was extended to a PDP-11 and partitioned into the same functions. For each function, appropriate classes of faults were defined. The same fault classes used by Abraham for the register decode, data storage, and data transfer functions were used in RTI's model. The fault class for the instruction decode and control function was modified. Since Abraham did not consider ALU faults, due to the variety and diversity of data

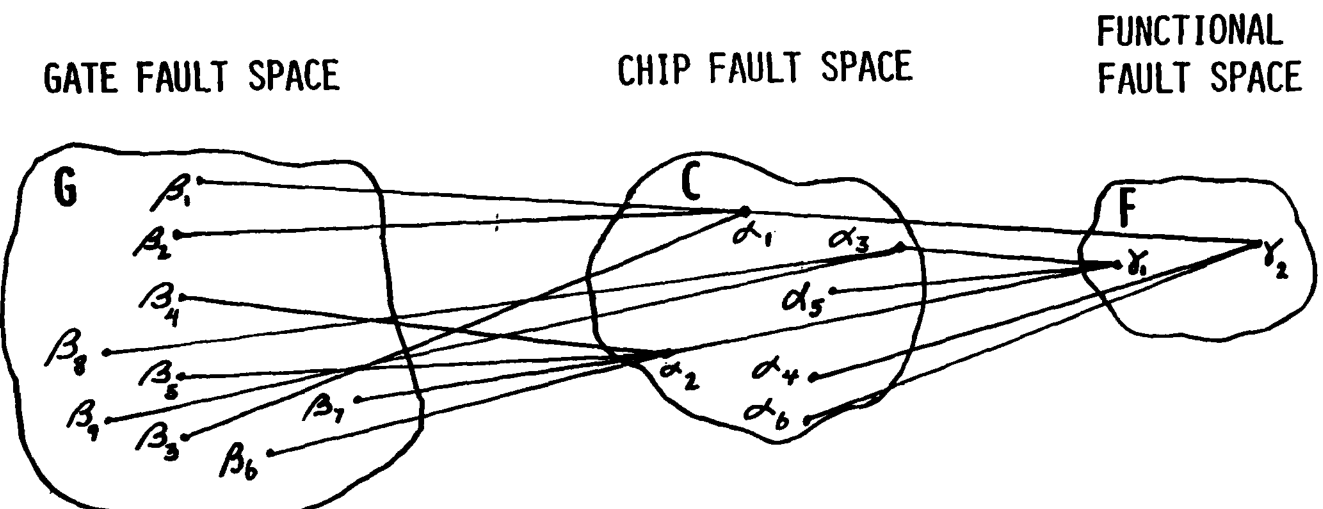


Figure 13. Digital Systems Fault Propagation.

manipulation functions, it was necessary to define a fault class for the ALU module. The five functions of interest in the PDP-11 system are thus summarized as follows:

1. register decode function,
2. instruction decode and control function,
3. data storage function,
4. data transfer function,
5. data manipulation (ALU) function.

(1) Register Decode Function

One or two registers may be associated with each instruction in the PDP-11. The following types of functional faults may occur in decoding these registers:

1. No register is selected.
2. The wrong register is selected.
3. More than one register is selected (inclusive or exclusive of the correct register).

(2) Instruction Decode and Control Function

The PDP-11 instruction format [7] can be one-word long (CLR R_0), two-words long (ADD #2,R_0), or three-words long (ADD #2,A). The types of functional faults that may occur in the instruction decode and control function are:

1. $I_j/\emptyset$

   Instead of executing instruction I_j, no instruction is executed.

2. I_j/NONE

   Since all of the codes in the PDP-11 family are not assigned to an operation (instruction), it is possible that instead of executing instruction I_j, a nonexistent instruction is selected.

3. I_j/undetermined

   Instruction I_j is incorrectly executed, not necessarily as another single instruction, but possibly as a combination of instructions. This is primarily due to the microcode (if microcoded), not decoding. This is a very general case which can

include any of the other mentioned cases. A system should be designed so that there is not such an occurrence.

4. I_j/I_k (length of $I_j \leq$ length of I_k)

   Instruction I_k is executed instead of I_j. For example:

   A 1-word instruction is executed as another 1-word instruction.
   A 1-word instruction is executed as a 2-word instruction.
   A 1-word instruction is executed as a 3-word instruction.
   A 2-word instruction is executed as another 2-word instruction.
   A 2-word instruction is executed as a 3-word instruction.
   A 3-word instruction is executed as another 3-word instruction.

5. I_j/I_j+I_k

   Instruction I_j is executed but another instruction is also executed. An example is the case in which a microsequencer fails at the end of an instruction and the next instruction, I_k, occurs immediately after the end of the current instruction, I_j.

6. I_j/I_M+I_N (length of $I_j \leq$ length of I_M+length of I_N)

   In this case one instruction, which is 2- or 3-words long, is executed as two or more instructions. A 2-word instruction may be executed as two 1-word instructions, a 1-word and a 2-word instruction, or as a 1-word and a 3-word instruction.

The following table shows some of the possible combinations that may occur in this case:

2w ⟶	1w and 1w
	1w and 2w
	1w and 3w
	1w and None
	1w and Ø
3w ⟶	1w and 1w and 1w
	1w and 1w and 2w
	1w and 1w and 3w
	1w and 1w and None
	1w and 1w and Ø
	2w and 1w
	2w and 2w
	2w and 3w
	2w and None
	2w and Ø

(3) Data Storage Function

Data storage is associated mostly with the registers in the machine. The types of faults that may occur are the classical stuck-at faults; i.e., one or more bits of a register are stuck-at-zero or stuck-at-one.

(4) Data Transfer Function

Each instruction is associated with a number of data paths. The faults that may occur are stuck-at faults.

(5) Data Manipulation Function (ALU)

The methodology used in this study was to make educated guesses about the different classes of functional faults in an ALU. Beginning with hypotheses for ALU faults that seem fairly reasonable, validation was accomplished by simulating an ALU at the gate level and observing different classes of faults. The remainder of this section discusses the hypotheses considered, the simulation process used to validate these hypotheses, and the results of the simulation.

(a) Hypotheses. Ten classes of functional faults were hypothesized for the TI 74181 ALU [8].* They are presented below, along with examples for each class.

1. No operation is performed. In this case the ALU is not operational; e.g., there is no power.
2. The wrong operation is performed. This class would appear to be a general class to include all the other fault classes. However, this is not true. What is included are the cases where an unknown operation is performed; e.g., if it is not known if A plus B changes to A minus B or if the results are all positive or negative. Usually, this is a class where the result is wrong and it is an unknown wrong.
3. Part of the operation is performed. The operation A plus B is performed, but the condition codes, such as the carry bit code and the overflow bit code, are not set.

*For more detailed information on the 74181 ALU, refer to Appendix B.

4. The complement of the operation is performed. In this case, the ALU may subtract two numbers rather than add them, or instead of rotating the operand to the right, it rotates the operand to left.
5. The operation is performed on the complement of the operand. Here, instead of adding B to A, the ALU adds -B to A.
6. The operation is performed with one variable always a constant. One leg of the ALU is stuck at a constant value.
7. Only positive or negative results are obtained. The sign bit code is stuck-at-zero or stuck-at-one.
8. A change from an arithmetic to a logical operation occurs or vice versa. This can be a general case where one ALU function changes to another. For example, A plus B changes to $A \oplus B$ or A plus 1 changes to A.
9. The result is $\pm$ a constant. In this case the final result (faulted result) is always equal to the correct result $\pm$ a constant.
10. Any number of output bits are stuck-at-zero or stuck-at-one. At first appearance this class might also appear to be inclusive of all the other classes discussed. However, this class includes only those faults which have not been mentioned before, for which one or more of the output bits are stuck at a value or are stuck at different values.

In order to study the ten classes of faults presented here, a small gate-level simulator was written and a 4-bit slice of a TI 74181 ALU was simulated. Two types of gate-level faults were injected; namely, stuck-at-zero and stuck-at-one.

The first simulation pass was not very detailed; only a handful of gates were faulted. Two major groups of faults were identified, regular and irregular. Regular faults are those which, for a specific function and all combinations of inputs to the ALU, there is a unique relationship between the faulted result and the correct result or inputs to the ALU.

With irregular faults, for a specific function and all combinations of input to the ALU, there is no unique relationship between the faulted result and the correct results or inputs.

(b) Experimental Approach. A more detailed simulation followed the initial simulation. Faults were injected in each gate of the TI 74181 ALU, both for the 30 instructions used in the PDP-11/70 (Table 2) and for all combinations of inputs for a 4-bit slice of the ALU.

For any combination of inputs, a fault at the gate level (stuck-at fault) generally manifests itself at a functional level, or it will not manifest itself at all. In the latter case, the function is independent of specific gate fault and is thus called a gate-independent function.

The relationship between a gate and a function is shown in Figure 14. The definitions of the terms used in Figure 14 are as follows:

1. Gate-independent: The specific function is independent of a specific gate; i.e., the specific function will not be in error if the specific gate is either stuck-at-zero or stuck-at-one, for all combinations of inputs.
2. Gate-dependent: The specific function is in error for at least one combination of inputs when the specific gate is faulted: the gate is either stuck-at-zero or stuck-at-one.
3. Zero- or one-sensitive: A specific function is zero-sensitive or one-sensitive with respect to a specific gate. The specific function is zero- (one-) sensitive if for at least one combination of inputs, the function is in error when the specific gate is stuck-at-zero (one), and if for all combinations of input, the specific function is not in error when the specific gate is stuck-at-one (zero).
4. Zero- and one-sensitive: A specific function is zero- and one-sensitive with respect to a specific gate. The specific function is zero- and one-sensitive if it satisfies the following two conditions:

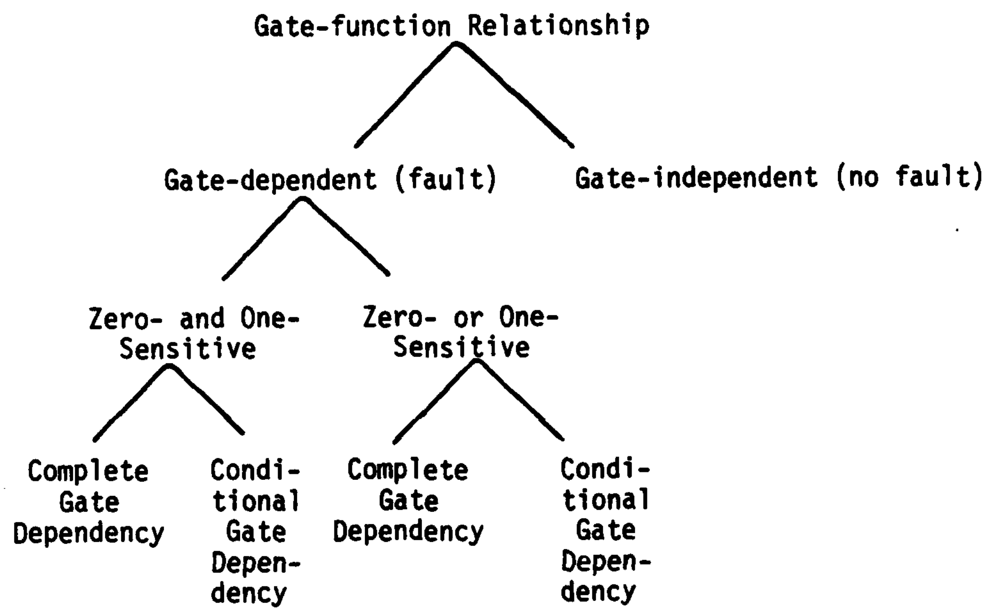


Figure 14. The Relationship Between a Function and a Gate in the TI 74181 ALU.

1) The function is in error for at least one combination of inputs when the gate is stuck-at-zero and the gate's true output value should be one.

2) The function is in error for at least one combination of inputs when the gate is stuck-at-one and the gate's true output value should be zero. This input combination does not necessarily have to be the same as 1).

5. Conditional gate dependency: For some, but not all, input combinations the specific function will be in error if the specific gate is stuck-at-zero (one) and the gate's true output should be one (zero). The condition on this gate's dependency is the input pattern. Notice that Figure 14 describes the relationship between a specific function and a specific gate and not the value that the gate is stuck at.

6. Complete gate dependency: The specific function will always be in error if the specific gate is stuck-at-zero (one) and the gate's true output should be one (zero).

If the gate fault manifests itself at the functional level (i.e., the function is not gate-independent), then there is a functional fault. Simulation of the TI 74181 ALU demonstrated the following general classes of functional fault classes as illustrated in Figure 15. Initially, there are two major classes of functional faults:

1. Functional Faults Independent of Inputs to the ALU: The specific function will always be in error for all combinations of input when a specific gate is stuck at a specific value.

2. Functional Faults Dependent on the Input to the ALU: The specific function will be in error for some, but not all, combinations of input when a specific gate is stuck at a specific value.

Notice that both of these classes apply to the gate number and the stuck-at value of the gate; i.e., for cases in which a gate is only stuck-at-zero or only stuck-at-one; not for cases in which a gate is stuck-at-zero and stuck-at-one.

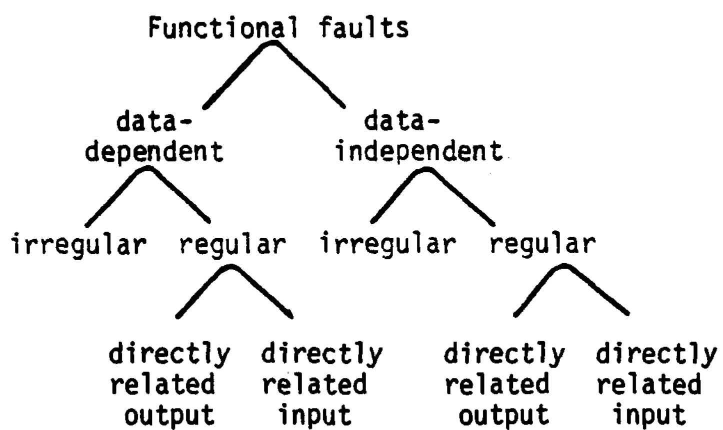


Figure 15. Classes of Faults in the TI 74181 ALU Simulation.

Each of the two major classes of functional faults discussed above can be further divided into two subclasses: namely, regular and irregular faults.

1. Regular Faults: The regular subclass of faults is that for which, given a specific function and a specific gate stuck at a specific value, the final result (faulted result) has a consistent relationship with the correct result or the inputs to the ALU for all combinations of inputs for which the specific function is in error. For example, the faulted result is the correct result - 4 for all values of the inputs that fault the ALU, or the faulted result is the result of a wrong operation on the inputs to the ALU. In other words, A plus* B changes to A + B.

2. Irregular Faults: The irregular subclass of faults is that for which, given a specific function and a specific gate stuck at a specific value, there is no consistent relationship between the faulted result and the correct result or the inputs to the ALU for all combinations of inputs for which the specific function is in error. For example, for one set of inputs the faulted result is the correct result - 4; for another set of inputs the faulted result is the correct result + 4. For one set of inputs A plus B changes to A + B, and for another set, A plus B changes to $\overline{A \oplus B}$.

To continue the breakdown of functional faults in Figure 15, the regular class divides into two subclasses. These are directly related input and directly related output faults.

1. Directly Related Input Faults: In this category of faults there is no straightforward relationship between the faulted result and the true result of a specific function when a specific gate is stuck at a specific value, but there is a relationship between the faulted result and the inputs to the ALU; e.g., A PLUS B changes to A exclusive ORed B.

2. Directly Related Output Faults: In this category of faults there is a straightforward relationship between the faulted result and the correct result of a specific function when a specific gate is stuck at a specific value; e.g., faulted result is equal to the correct result exclusive ORed with a constant m.

---

*This notation is used by the TI ALU 74181 specification. Here, '+' refers to the logical 'OR', 'plus' addition, etc. For additional information, refer to Appendix B.

In summation, it should be noted that the gate-function relationship (Figure 14) is presented in a two-dimensional concept, gate and function; it is a breakdown, in essence, of the types of ALU functions. The classification of faults in the ALU simulation (Figure 15) is presented as a node of the fault tree (gate-dependent) in a three-dimensional space, gate, function, and the stuck-at value. The fault tree is a breakdown of the types of faults that can occur.

(c) Measurements. Several different measurements were performed using the ALU simulator.

1. Fault populations. It should be noted that there are 62 gates, 30 functions, and 256 different combinations of inputs in the ALU. If for every combination of inputs each fault caused each of the 30 functions to be in error (i.e., if each function was completely gate-dependent), the total number of faults possible would be 62*30*256 = 476,160. When any of the gates is stuck-at-zero, one at a time, the total number of functional (pin) faults is 141,046; when any of the gates is stuck-at-one, one at a time, the total number of functional (pin) faults is 221,097. When any of the gates is stuck-at-zero or stuck-at-one, one at a time, the total number of faults is the sum of these two, or 362,143.

Measurements were completed for all combinations of inputs (i.e., 256 for a 4-bit slice of the TI 74181 ALU), with a gate-level fault manifesting as a functional fault if any of the 7 pins (i.e., $\overline{X}$, $\overline{Y}$, $\overline{C}$, F_3, F_2, F_1, F_0) were in error. The total number of faults that actually occurred were 362,143, which is 76% of the total possible faults.

One interesting point in these measurements is the difference between the number of faults occurring when gates are stuck-at-zero (39% of the total) and the number of faults occurring when gates are stuck-at-one (61% of the total). This difference is due to the number of AND gates used in the TI 74181 ALU. If a three-input AND gate is used with the probability that each input has an equal chance to be zero as well as one, the probability of the output being one is 1/2*1/2*1/2 = 1/8, and the probability of the output being zero is 1 - 1/8 = 7/8. It is obvious that stuck-at-one

will hurt more than stuck-at-zero. The reverse is true for a circuit with more OR gates than AND gates.

2. Effect of gate-level faults on ALU functions. Tables 4, 5, and 6 indicate what effect gate-level faults have on ALU functions. The frequency with which a functional fault occurs and the percentage of the time a fault occurs, with respect to the total number of functional faults, are presented for cases when the gates are stuck-at-zero, stuck-at-one, or stuck-at-zero or stuck-at-one. (The gate's value is always incorrect). These tables are arranged according to the function affected most often. Figure 16 is a histogram of the results sorted for all the gates stuck-at-zero or stuck-at-one, one at a time. As indicated, there is little difference between the function most affected and the function least affected (only 0.72%). Figures 17 and 18 show the corresponding histograms when all the gates are stuck-at-zero or stuck-at-one, respectively, one at a time. These two histograms are not arranged in descending order. Note that there is little difference between the function most affected and the function least affected for either case.

Even though there is little difference in the frequency with which each instruction will be affected, the frequency with which an instruction (ALU function) is used is a big factor in specifying which instruction will be most affected. Table 7, extracted from Edward Snow's work [15], shows the frequency of execution of some instructions in the PDP-11 family. The probability that an instruction is in error is equal to the cross product of the probability that that specific instruction gets executed and the probability that there is a fault in executing that instruction. Table 8 presents calculations completed for some typical PDP-11 instructions. As shown, the sequence of most likely instructions in error has changed. Also, keep in mind that these results are obtained over all possible combinations of inputs. The input data distribution is also an important factor.

3. Frequency of occurrence of different types of functional faults. The TI 74181 ALU simulation was exercised to measure the frequency of regular and irregular faults (data-dependent and data-independent). For

Table 4. Frequency and Percentage of Times a TI 74181 ALU Function Will Be in Error When a Gate Is Stuck-at-Zero.

PDP-11/70 ALU* Instruction	Function	Number of Times In Error	Percent with Respect to Total Number of Errors
	Minus 1 (2's complement)	5376	1.48
	$A\overline{B}$ minus 1	5133	1.41
SUB	A minus B	5095	1.40
BIS	A+B	5077	1.40
	$(A+\overline{B})$ plus A plus 1	5071	1.40
	1 (logical)	5040	1.39
XOR	$A \oplus B$	4995	1.37
	AB minus 1	4877	1.34
MOV	A	4848	1.33
DEC	A minus 1	4848	1.33
MOV	A	4848	1.33
	$(A+\overline{B})$ plus AB plus 1	4839	1.33
BIC	$A\overline{B}$	4783	1.32
BIC	$A\overline{B}$	4783	1.32
	A minus B minus 1	4771	1.31
	A plus A plus 1	4752	1.31
MOV	B	4739	1.30
	$(A+\overline{B})$ plus A	4645	1.28
	A plus B plus 1	4583	1.26
BIT	AB	4527	1.25
BIT	AB	4527	1.25
	$(A+\overline{B})$ plus AB	4515	1.24
	$\overline{A \oplus B}$	4483	1.23
INC	A plus 1	4464	1.23
COM	$\overline{A}$	4432	1.22
CLR	Zero	4352	1.20
CLR	0 (logical)	4352	1.20
ADD	A plus B	4259	1.17
ASH	A plus A	4144	1.14
MOV	A	3888	1.07
TOTAL		141046	31.0%

*Note that there may be alternative ways to realize the PDP-11/70 instruction with the 74181.

Table 5. Frequency and Percentage of Times a TI 74181 ALU Function Will Be in Error When a Gate is Stuck-at-One.

PDP-11/70 ALU Instruction	Function	Number of Times In Error	Percent with Respect to Total Number of Errors
COM	$\overline{A}$	8224	2.27
XOR	$A \oplus B$	8197	2.26
MOV	B	7941	2.19
CLR	Zero	7936	2.19
CLR	0	7936	2.19
	1 (logical)	7920	2.18
	A plus A	7824	2.16
BIC	$A\overline{B}$	7761	2.14
BIC	$A\overline{B}$	7761	2.14
BIS	A+B	7760	2.14
	$\overline{A \oplus B}$	7685	2.12
	$(A+\overline{B})$ plus A	7568	2.08
BIT	AB	7505	2.07
BIT	AB	7505	2.07
SUB	A minus B	7457	2.05
	A minus B minus 1	7361	2.03
	A plus A plus 1	7312	2.01
	$(A+\overline{B})$ plus A plus 1	7250	2.00
	$(A+\overline{B})$ plus AB plus 1	7201	1.98
MOV	A	7184	1.98
MOV	A	7184	1.98
	$(A+\overline{B})$ plus AB	7105	1.96
DEC	A minus 1	7088	1.95
	A plus B plus 1	6945	1.91
ADD	A plus B	6849	1.89
	$A\overline{B}$ minus 1	6863	1.89
INC	A plus 1	6816	1.88
MOV	A	6720	1.85
	AB minus 1	6607	1.82
	Minus 1 (2's complement)	5632	1.55
TOTAL		221097	69.0%

Table 6. Frequency and Percentage of Times a TI 74181 ALU Function Will Be in Error When a Gate is Either Stuck-at-Zero or One.

PDP-11/70 ALU Instruction	Function	Number of Times in Error	Percent with Respect to Total Number of Errors
XOR	$A \oplus B$	13192	3.64
	1 (logical)	12960	3.57
BIS	A+B	12837	3.54
MOV	B	12680	3.50
COM	$\overline{A}$	12656	3.49
SUB	A minus B	12552	3.46
BIC	$A\overline{B}$	12544	3.46
BIC	$A\overline{B}$	12544	3.46
	$(A+\overline{B})$ plus A plus 1	12321	3.40
CLR	Zero	12288	3.39
CLR	0	12288	3.39
	$(A+\overline{B})$ plus A	12213	3.37
	$\overline{A \oplus B}$	12168	3.35
	A minus B minus 1	12132	3.35
	A plus A plus 1	12064	3.33
	$(A+\overline{B})$ plus AB plus 1	12040	3.32
BIT	AB	12032	3.32
BIT	AB	12032	3.32
MOV	A	12032	3.32
MOV	A	12032	3.32
	$A\overline{B}$ minus 1	11996	3.31
	A plus A	11968	3.30
DEC	A minus 1	11936	3.29
	$(A+\overline{B})$ plus AB	11620	3.20
	A plus B plus 1	11528	3.18
	AB minus 1	11484	3.17
INC	A plus 1	11280	3.11
ADD	A plus B	11108	3.06
	Minus 1 (2's complement)	11008	3.03
MOV	A	10608	2.92
TOTAL		362143	100.0%

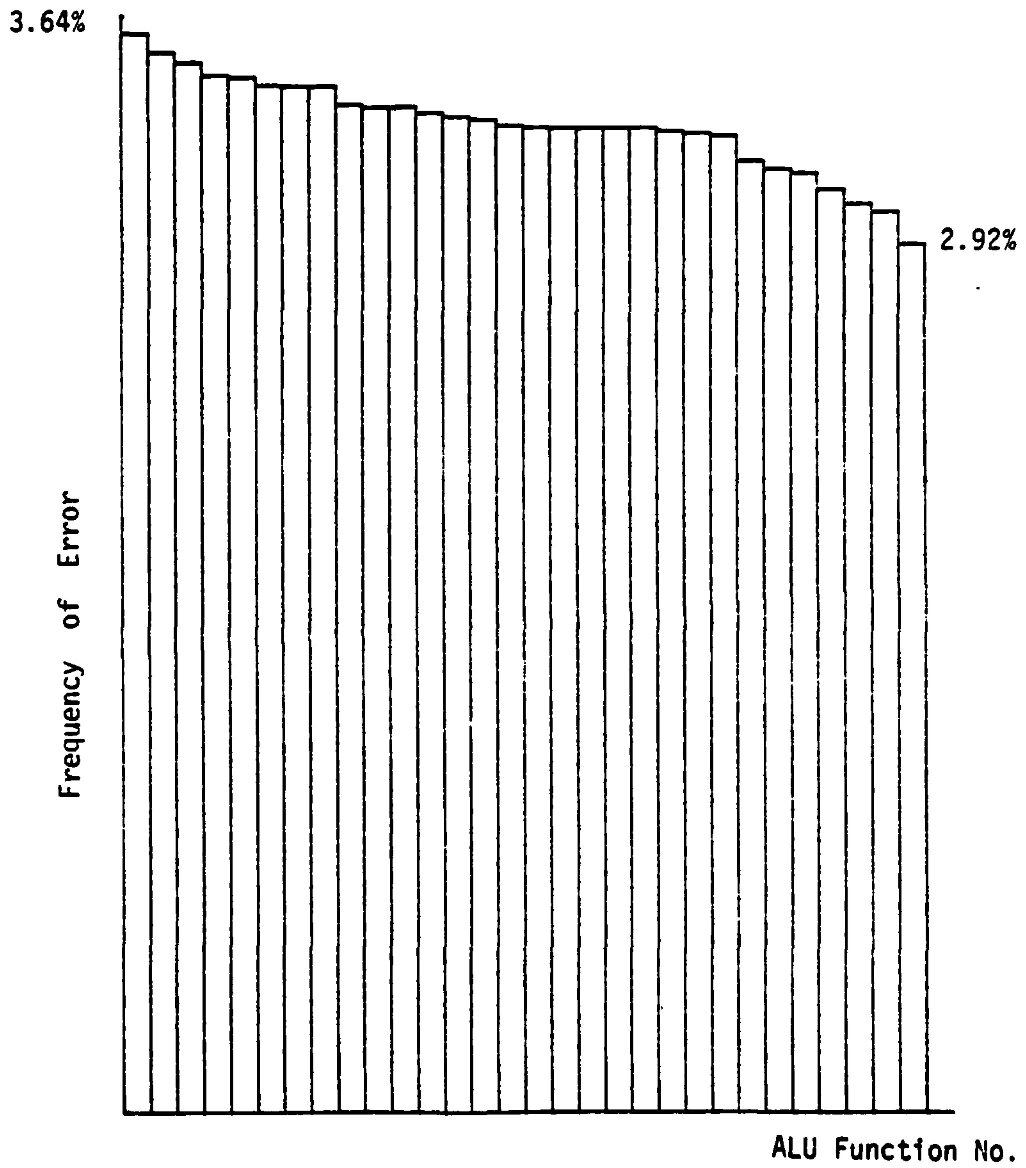


Figure 16. Frequency of an ALU Function in Error When All the Gates Are Stuck-at-Zero or Stuck-at-One, One at a Time.

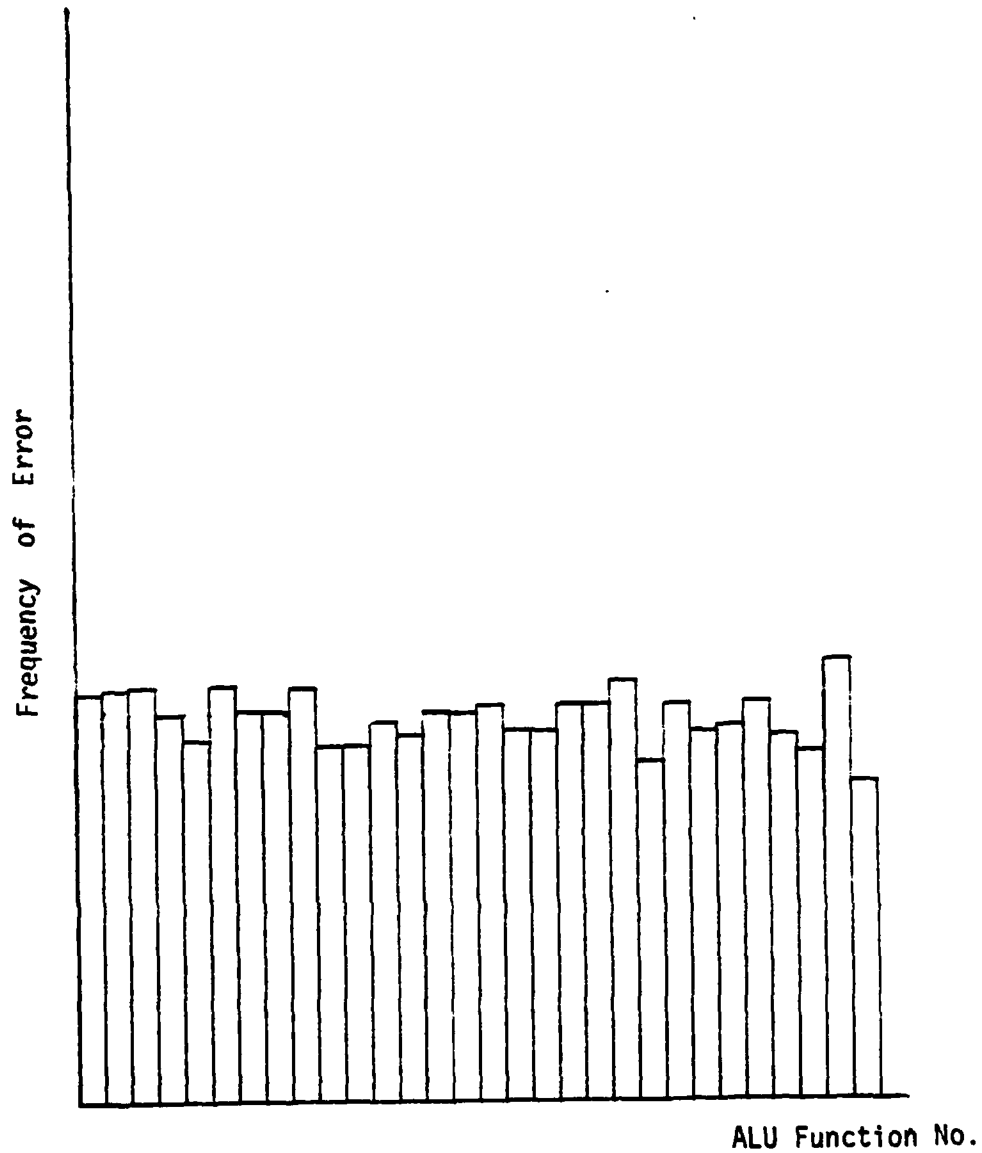


Figure 17. Frequency of an ALU Function in Error When All the Gates Are Stuck-at-Zero, One at a Time.

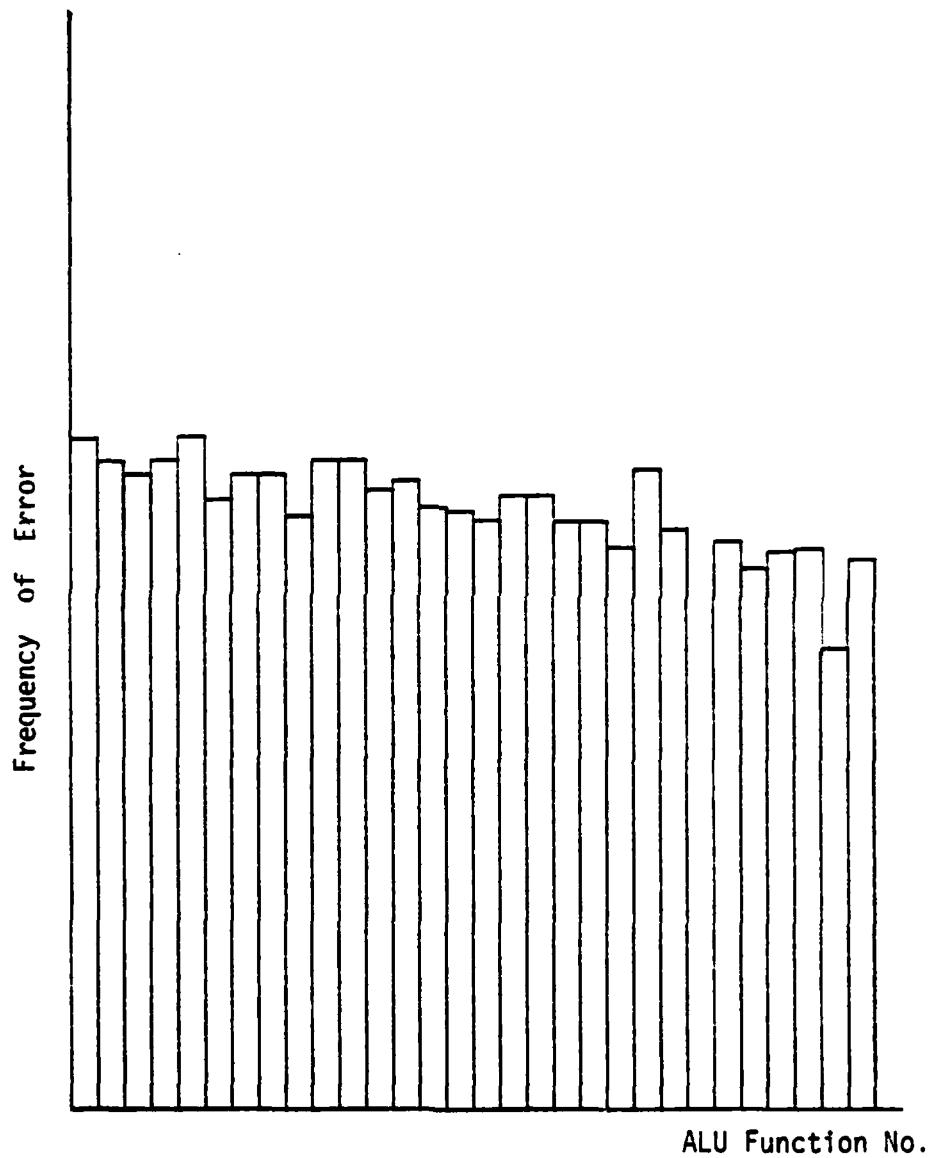


Figure 18. Frequency of an ALU Function in Error When All the Gates Are Stuck-at-One, One at a Time.

Table 7. Some Typical Instruction Execution Frequencies.*

XOR	<0.0005
COM	<0.0005
BIS	0.0012
BIT	0.0041
CLR	0.0186
INC	0.0224
SUB	0.0274
BIC	0.0309
ADD	0.0524
DEC	0.0809
MOV	0.1517

* For typical PDP-11 instructions which use 74181.

Table 8. Frequency of Error for Typical PDP-11 Instructions.**

PDP-11 Instruction	Probability of Error	ALU Function Used
COM	0.0349*0.0005 = 0.000017	$\overline{A}$
XOR	0.0364*0.0005 = 0.000018	$A \oplus B$
BIS	0.0354*0.0012 = 0.000042	A + B
BIT	0.0332*0.0041 = 0.000136	AB
INC	0.0311*0.0224 = 0.000606	A plus 1
CLR	0.0339*0.0186 = 0.000630	zero
SUB	0.0346*0.0274 = 0.000959	A minus B
BIC	0.0346*0.0309 = 0.001069	$A\overline{B}$
ADD	0.0306*0.0542 = 0.001603	A plus B
DEC	0.0329*0.0809 = 0.002661	A minus 1
MOV	0.0332*0.1517 = 0.005036	A

**Assume 4-bit ALU results directly extended to 16-bit ALU.

all combinations of inputs to the ALU, there are 118,514 regular functional faults for any gate stuck-at-zero. Of this number, 116,194 are directly related output faults, 73,986 of which occur when the faulted result = correct result $\oplus$ m = correct result $\pm$ n. A total of 99,362 directly related output faults occur when the faulted result = correct result + m; this includes the previous category. Of the total 116,194 faults, 90,818 occur when the faulted result = correct result $\pm$ n; this includes the first class. An important result is that 70% of the faults (99,362/141,046) are single-pin faults since m is usually a 4-bit binary number with only one bit on. (This does not include faults in the $\bar{X}$, $\bar{Y}$ and $\bar{C}$ pins.) The measurements also show that 84% of the faults (118,514/141,046) are regular errors when all the gates are stuck-at-zero, one at a time.

Similar results are obtained when all the gates are stuck-at-one, one at a time. There are 175,990 regular faults of which 174,950 are directly related output faults. Of the directly related output faults, 109,510 occur where the faulted result = correct result + m = correct result $\pm$ n. A total of 156,006 faults occur when the faulted result = correct result $\oplus$ m, including the first case. A total of 128,454 faults occur when the faulted result = correct result $\pm$ n, including the first case. Again, 70% of the faults (156,006/221,097) are single-pin faults and 79% (175,990/221,097) are regular faults.

Perhaps a more accurate measurement can be considered in terms of cross products of the gates and functions. There are 62 gates in the ALU (not including the gate for equality). Thirty out of 48 functions are used by the PDP-11/70. Therefore, 1860 (62*30) functional faults can occur when a gate is stuck-at-zero or stuck-at-one. Eighteen of the 30 functions available are independent of one or more gates stuck-at-zero or stuck-at-one. Of the total functional faults (1860), 1049 faults occur when all of the gates are stuck-at-zero, one at a time; 33 of which are irregular. The remainder (1016) are regular faults. When all of the gates are stuck-at-one, one at a time, a total of 1,255 faults occur. Of these, 74 are irregular and 1,181 are regular. Figure 19 presents the exact breakdown for these conditions.

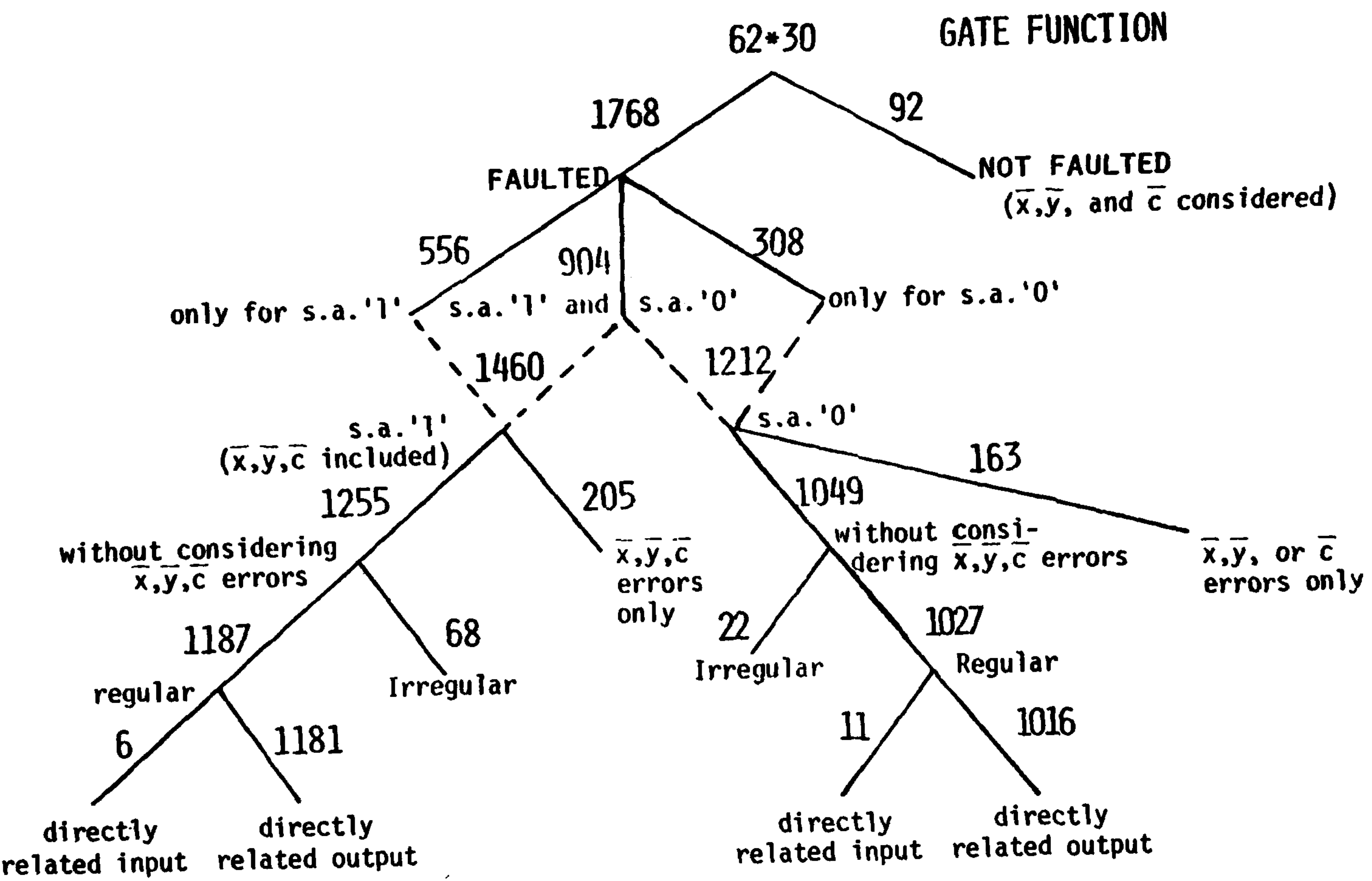


Figure 19. Statistics Expressing the Relationship Between Functions and Gates in the 74181 ALU.

4. Refined functional fault model. At this point it is useful to compare the hypotheses originally stated and the experimental results concerning types of ALU faults. The results of the simulation produced the following types of faults:

1) Faulted result = correct result $\oplus$ m

   or

   Faulted result = Correct result $\pm$ n where m is a binary number with one or more bits on.

   In the case of the TI 74181 ALU one bit is generally on, and n is an exponent of 2 (i.e., 2^1, 2^2, 2^3) for 4-bit slice. It should be noted that the results obtained for a 4-bit slice are applicable to cases other than a 4-bit ALU. The only interaction between the 4 least significant bits and the next 4 bits is the carry bit (or $\overline{X}$ and $\overline{Y}$). This would only change the m and n numbers; in other words, m will be a 16-bit number with more than 1 bit on, and n will be a polynomial in the form $a_1 2^{k-1} + a_2 2^{k-2} + \ldots + a_k 2^1$.

2) Change of operation.

   In this case, the logical operation changes to an arithmetic operation or vice versa. Generally, any operation can change to another operation.

3) Any of the output bits, including carry (C), carry generate (X), and carry propagate (Y), can be stuck-at-zero, stuck-at-one or flipped.

Comparing the ten classes of functional faults originally hypothesized with the experimental results of the simulation, it is evident that fault types in classes 1 and 6 are eliminated. Classes 2, 4, 5 and 8 fall into category b of the experimental results. Fault types in classes 2, 7 and 10 fall into category c of the experimental results, while faults in class 9 fall into category a of the experimental results.

4. FUNCTIONAL FAULT INJECTION

a. Introduction

This chapter is devoted to the injection of the functional faults predicted in the previous chapter into the PDP-11/70 ISP description described in Chapter 2.

Functional fault injection was performed on two different systems -- a PDP-11/70 without any built-in tests (the single CPU discussed in Chapter 2) and two PDP-11/70 CPUs (a dual CPU) functioning in parallel with their memory busses compared upon each memory reference. Two main objectives were involved in selecting these two systems. The first objective was to observe the manifestation of functional faults at the system level; i.e., system behavior in a faulty environment. Using this information, a system with appropriate built-in tests can be designed to catch most of the faults. The second objective was to study a system with a built-in test to observe detection coverage and system behavior with a BIT present.

The rest of this chapter will discuss the programs selected for fault injection, the different parameters measured, and the results obtained from functional fault injection in each of the functional modules in the single- and dual-CPU systems.

b. Strategy

The following sections describe the programs selected for functional fault injection. Parameters to be measured are also explained.

(1) Program Selection

Two options were considered for program selection: writing a new set of programs or choosing a set already written. Since a new set of programs would have required a large expenditure of effort to choose the application area and to write the programs, a set of programs were selected that had already been written and tested for use in benchmarking the ISP descriptions of the CFA [3,16] program.

Sixteen different program categories are available for selection. They are listed in Table 9 [16]. (Each program was written by two different programmers.) Seven of these appeared applicable; the remainder

used instructions such as floating point and extended instruction set, which have not been implemented in the RTI version of the PDP-11/70 ISP description.

As explained further in the next section, several measurements were worth studying. However, after the initial phase of fault injection, it was obvious that it was not possible to measure all of these parameters. Depending on the environment, the fault latency would have varied significantly. Therefore, it was not feasible to concentrate on the frequency and latency of each class of errors. Instead, attention was directed to how each class of faults manifested itself at the system level. Two programs (12, 13) which use most of the PDP-11/70 instructions and used the most frequently occurring instructions were chosen. These two programs are presented in Appendix C(1).

Table 9. CFA Programs.

Interrupts and traps
(0) terminal input driver
(1) message buffering and transmission
(2) multiple priority interrupt handler
(3) virtual memory exchange

Miscellaneous
(4) scale vector display
(5) array manipulation-LU decomposition
(6) target tracking
(7) digital communications processing

Address manipulation
(8) hash table search
(9) linked list insertion
(10) presort (over a large address space)
(11) autocorrelate (over a large address space)

Character and bit manipulation
(12) character search
(13) boolean matrix transpose
(14) record unpacking
(15) vector to scan line conversion

(2) Measurements

Initially, three types of parameters looked promising for observation and measurement: 1) the classes of faults manifested at the system level,

2) the frequency of occurrence of each of these classes, and 3) the detection latency of these faults. After the first phase of fault injection, it was seen that almost all of the manifestations classified were interchangeable, depending upon the program and the environment in which the program was running. An accurate frequency of occurrence could not be measured because any frequency of occurrence measured would have been for a specific environment, a specific state, and a specific program. Furthermore, the error latency could not be correctly measured because it also would have referred to a specific environment,* a specific program, and a specific state.** Thus, the only parameter that could be measured after the first phase of fault injection was the classification of faults at the system level.

c. Single-CPU Fault Injection

In this section, functional fault injection for the single CPU is discussed in terms of each of the four function modules: register decode, instruction decode, data storage, and data manipulation (ALU). Fault injection for the data transfer function module is not discussed because this class of fault injection is highly dependent upon the bus structure. In addition, some of the manifestations for data transfer module will manifest themselves when data storage modules are injected with faults. Functional fault injections for all single-CPU functions generally manifest themselves as stuck-at-zero and stuck-at-one faults. Consequently, the results represent stuck-at-zero and stuck-at-one at the gate level. Whenever possible, the ISP simulator was used for fault injection. In other situations, fault injection was performed manually, or the ISP description was modified to accommodate the necessary hooks for a better and faster fault injection. These modifications will be discussed later.

---

*Environment refers to the location where the program is loaded into memory, the operating system support, the content of the interrupt vector locations (are they loaded or not), (interrupt handlers), and any other user states.

**System states refers to the contents of various architectural registers (used and unused by the program) and to the contents of various implementation registers (not an architectural feature and not seen or used by the programmer).

(1) Register Decode

Register files are a ubiquitous module in most modern computers; they are most often directly visible at the microcode level. Generally, the programmer will only see a register file at the instruction set level when he uses register instructions. Since the microcode level is extremely implementation dependent and since the RTI model is viewed basically at the instruction set level, examples used will consist of injecting functional faults into a register file at the instruction set level. It should be noted that functional faults for the register file are the same, regardless of the level of the register file in the particular machine.

RTI's functional fault model of the register file is based on the functional block diagram presented in Figure 6. All data that is read or written to the register file is first gated into the register file buffer. The register to be read/written is selected by decoding the register index which activates one (and only one) select line. The model is covered by two fault classes, register decode and data storage. (See Section 4c(3))

Register decode functional faults are divided into three categories: 1) the wrong register is selected, 2) no register is selected, and 3) multiple registers are selected. Each category is discussed below in terms of what the category means, how it is physically realizable, how it is executed in the simulator, and what are the simulation results.

1) The wrong register is selected.

For this category a wrong select line is active, but only one select line is active. As a result, a wrong register is selected. This may occur if an input or select line is stuck-at or if one of the internal gates is faulted. In the simulator, the fault injection mechanism of ISP [12] is used to fault the input bits (i.e., the index to the register file) to the decoder.

Table 10 tabulates the simulation results for this category. The stuck-at-zero input bit faults all produced the same results. Why? Generally, when a program is loaded into memory for execution, all the general-purpose registers are set to zero (the initialization process of

the ISP simulator) except the R7(PC), which points to the starting address of the program. Under stuck-at-zero faults, R7 can never be accessed. Its address is binary 111; thus, whenever another register is accessed in its place (R6, R5 or R3), the contents of the second register will be used as the next instruction address. If the contents of these registers are the same, then the results will be the same.

Table 10. Wrong Register Is Selected.

Register File Index Is Stuck-At	Results	Comments
XX0	data executed	*Instruction address pointed out of program bounds; address was in system data area. HALT executed.
X0X	"	
0XX	"	
XX1	data executed	*Changes to SP modified PC. PC pointed to program data. WAIT executed.
X1X	"	*Changes to R5 modified PC PC points to program data.
1XX	"	*MOV 11(r6),020760(r4) executed causing odd address error.
1X1	"	*Same as XX1 case.
11X	odd address error	*Changes to R1 modified PC making PC odd.

X = do not care

For the stuck-at-one cases the PC was modified, in an attempt to modify another register and to point out of the instruction boundaries of the program and resulted in data being executed. In one case the PC modification resulted in an odd value, which caused an odd-address trap. An example of the stuck-at-one case follows:

Bit 2 of the register index is stuck-at-one; 1XX

```
NEWROW: ADD R4,R0        => ADD R4,R4       ; R0 is changed to R4
        CMP R0,#TOPMASK  => CMP R4,#TOPMASK ; R0 CHANGED TO R4
        BLT 1$                              ; branch taken
            .
            .
            .
1$    : MOV R4,R5                           ; no changes
        MOV R2,R3        => MOV R6,R7       ; R2 changed to R6 and R3
                                            ; changed to R7
```

This caused R7 to point into the data segment of the program; as a result, the data was executed. The data was interpreted as

MOV 11(R6),020760(R0).

When executed, this resulted in an odd-address error because the index of 11 is odd and the instruction is a word instruction.

2) No register is selected.

For this category none of the select lines are active for certain input patterns. In this case, only the one select line is affected. Physically, this can be realized by faulting a select line output gate to the nonactive state. The question to be asked is what happens on a read/write request to the register file if none of the select lines are active. For the write request, the data to be written is loaded into the buffer and the write operation is suppressed. The only change is to the buffer. To read the file the contents of the buffer are used as the results of a read request. Whatever was last read or written into the register file buffer is taken as the data to be read. In the ISP simulator, this occurs by setting the NO.SEL bit in the register file code and setting the appropriate bit in REG.SEL which represents the no-register-selected case.

Table 11 tabulates the results of the no-register-selected case. Note that some very interesting side effects occurred. An instruction has been used as if it were data modified to be an entirely different, but legitimate, instruction that is executed. Trying to backtrack this type of error is very difficult, since the program continues to execute long after the

error occurs. Even with execution traces, it took considerable time to find out what had occurred.

Table 11. Register Is Not Selected.

Register Not Selected	Results	Comments
R0	infinite loop	*R0 is put on the stack. Suppose to be the array offset but in actuality the address of that instruction is put on the stack; instruction @SP was executed which modified the instruction from MOV (R0),-(SP) to CMP -(R1),@R4. This was added to a loop counter making it 25110. A very long loop. Once loop is completed, the modified instruction would be executed. But the program would continue with a very messed up stack.
R1	infinite loop	*R1 counts number of arrays to be transposed. When R1 is decremented by loop counter : DEC R1 the value returned to be decremented is the address of this instruction which will remain constant. Program will blow up when pointers to the arrays get fouled up.
R2,R4	HALT correctly	*Matrix transpose done in place. No change to original matrix. Does not mean that some other location in main memory was not changed.
R3,R5	HALT correctly	*Wrong result. The matrix was modified but not completely or correctly.
R7	data executed	*Executes instruction at whatever address was last in the register file buffer.
R6	stack trap	*In autodecrement mode SP is range checked against the stack limit register. Whatever was last in register file buffer is used as contents of SP.

For several registers the program correctly terminated, but the results of the computation were wrong. In two cases the transpose of the matrix was the same as the original matrix. In this instance, a careful

programmer might wonder if this transpose program were a subroutine called as a subelement in a larger program and might rerun the computation or check the results in some other manner. Two other programs that correctly terminated with incorrect results gave as a result a matrix that was different from the original. Unless the results were known or an extra check step was run, these wrong results might miss being detected.

These errors are very program dependent. If the program does not use the faulty register, then the problem will not be detected. If the register is used in an infrequently accessed segment of code, then again it might not be detected. Due to the way faults in this category manifest themselves, isolating the faults is extremely difficult. For this reason the category of register decode faults appears to be the most dangerous of all the faults.

In the following example, R6 will be the not-selected register. Remember in autodecrement and autodecrement-deferred addressing modes with SP as the register, SP is limit-checked for stack underflow.

```
BMT: MOV R0,-(SP) ; Contents of register file buffer is used as SP,
                  ; because SP is not selected. Buffer last changed when
                  ; R0 was accessed so SP=contents of R0=1434.

     MOV R1,-(SP) ; SP takes on register file buffer contents which is
                  ; the contents of R0=1, at this time. SP is first
                  ; decremented by two yielding a -1. This is checked
                  ; against the stack limit register, causing a
                  ; stack limit underflow trap.
```

3) Multiple registers are selected.

If multiple registers are selected, then multiple select lines must be active simultaneously. This occurs if several select line output gates are stuck-at-active. A number of questions arise in this case. What sort of outputs occur when reading the register? Are the multiple register ANDed or ORed together to produce a result? Is the desired register chosen along

with other undesired registers, or are only the unasked for registers chosen? These questions require consideration of four subcases They are: 1) the registers are ORed together with the desired register, 2) the registers are ANDed together with the desired register 3) the registers are ORed together without the desired register, and 4) the registers are ANDed together without the desired register. In the subcases lacking the desired register, if the desired register is among the set of stuck-at-active registers, then it will be used.

This category and its subcases are realized in the simulator by setting MULT.SEL (multiple registers selected), AND.OR (to choose the technology), INCL.SW (to include/exclude the desired register), and the appropriate bits in REG.SEL, which indicates which registers have their select lines permanently active.

Tables 12 and 13 tabulate the results when multiple registers are used simultaneously. Table 12 presents the subcase of the desired register being selected along with the registers that are always selected. For the OR technology the program executed several instructions, while for the AND technology only one instruction was executed. This again reflects the state of the registers when the fault occurs. If the registers had some previous values (nonzero), then several more instructions would have been executed for the AND technology.

Table 13 is the subcase of only the registers with select lines permanently active being used. Again, because the registers are zero, the PC is quickly forced out of the instruction bounds and it halts. In all the examples where PC is not one of the selected registers, this same outcome occurs. When PC is one of the active registers, it takes on whatever value is written to the register file and uses that value as the next instruction address. In RTI's experiment the PC took on the address meant for the SP and as a result, the processor executed the data at which the SP was pointing.

Table 12. Desired Register Is Included.

TECHNOLOGY			
OR	Register Selected	Results	Comments
	1,3,5	data executed	*PC modified; HALT executed.
	2,4,6	"	" " " "
	5,7	"	*PC forced to value of SP; WAIT executed.
	6,7	"	"" "" " "
AND			
	1,3,5	data executed	*PC modified forced to zero: pointed at system data area. This occurred because the contents of the registers were all set to zero when the program was loaded.
	2,4,6	"	
	5,7	" "	
	6,7	"	

Table 13. Desired Register Is Excluded.

TECHNOLOGY			
OR	Register Selected	Results	Comments
	1,3,5	data executed	*Contents of the registers were all zero. Forced the PC to zero resulting in data in system area being executed; HALT.
	2,4,6	"	
	5,7	"	PC was forced to the value of the SP as in part (a) of this table. WAIT was executed.
	6,7	"	
AND			
	1,3,5	data executed	*Registers were all zeros. PC was forced to zero; data from system was executed; HALT.
	2,4,6	"	
	5,7	"	" "
	6,7	"	" "

An example follows showing what happens when R5 and R7 are multiply selected. The technology is the OR type and the desired register is included in the operation.

```
Address          Instruction/Data
 01272           MOV #STACK,SP  ; #STACK is an address equal to 1424.
                                ; Putting address in SP also sets R5 and R7
                                ; to 1424 because they are always selected.

 01424  STACK: .WORD 1          ; Data is the number of matrices to
                                ; transpose. R7 points here as a result
                                ; of the previous MOV instruction.
                                ; The data is executed as a WAIT
                                ; instruction. The program halts.
```

In conclusion, after studying the execution traces of these examples, some general statements can be made. For the OR technology, registers affected tend towards higher values (i.e., more ones), while the reverse is true for the AND technology. This trend was demonstrated in the tables. When the program stopped, the PC was generally pointing into the low-memory locations for the AND technology and into the data segment of the program for the OR technology. In most programs, data locations are generally allocated after the instructions. In the PDP-11/70 the low-memory locations 0-1000(octal) generally hold trap addresses and other system-related data. Because no trap addresses or system information were stored in the low-memory addresses, the program halted (HALT=000000) when the processor tried to execute the values found there.

To detect these types of problems as they occur, the decoder could be designed to be self-testing. It could then signal an error if zero or more than one select line were active. This would then detect the most potentially dangerous fault, the one with the longest latency, the no-select category. This scheme, unfortunately, does not detect errors in the case where the wrong register is selected. To do that would require a duplicate decoder and comparing circuitry. The self-testing method could easily be incorporated into most existing register files. An additional line would

be required to signal when the self test failed. The duplicate decoder and comparator would not require any more additional lines than the self-test method, but it would require more chip real estate. However, both methods could be easily incorporated into existing architectures without any major changes.

Other methods would, in general, require changes to the architecture. For example, data and instructions could be placed in separate memory spaces. The instruction space could only be accessed while in the instruction fetch cycle, and the data space could only be accessed while in the execute cycle. It would be illegal to write into the instruction space, read only, and to execute data in the data space. These measures would detect the manifestations of the faults injected into the register file.

(2) Instruction Decode

PDP-11/70 op-codes are of variable length; the instructions can be one, two, or three words in length [7]. Op-codes can be from 4 bits to 16 bits long. Consequently, covering all possible cases of instruction decode, functional fault injection can be tedious and time-consuming, and it is essentially no more useful than selecting a set of representative cases for each class of faults.

The following pattern was followed throughout instruction decode fault injection. Since only permanent faults were of interest here, fault injection occurred for all instances of the same instruction whenever an instruction was faulted. Generally, the faults were injected before the program started; in other words, the faults were present throughout the execution of the program. In special cases faults were injected in the middle of program execution, and from that point on the fault existed.

During the fault injection process only one fault was injected at a time; multiple faults were avoided. For example, in one program run only instruction x was faulted; in another, instruction y, etc. In a 2- or 3-word instruction, for the class of faults where an instruction faulted to a no-operation instruction (NOP), only the first word was changed to a no-operation instruction; the second and third words were executed as part of the next instruction.

Appendix C(2) lists all the instruction decode functional faults which were inserted in the two CFA programs. In general, there were four end results from the simulations:

1) The program ends with the right result.

In this case, the faulted instruction is in a portion of the code which is never executed, or is in a branch statement which is never satisfied. Consider the following portion of the code from 0412 (one of the CFA programs for string search).

```
MATCH:  DEC     R5
        BLE     FOUND
        CMPB    (R4)+,(R3)+
        BEQ     MATCH
        BR      NOMATCH         !never executed
```

For the specific data used, "BEQ   MATCH" is always satisfied and "BR   NOMATCH" is never executed. It should be noted that there is no other BR statement in this program.

2) The program ends with a wrong result.

It should be clarified that program ends, refers to the fact that the program either stops at the point where it should have stopped in the no-fault case or it stops at some random point. This is also referred to as the halt case. For example, if a fault causes a branch instruction to change to a no-operation instruction or to another branch instruction with a different destination address, still within the boundary of the program, the following (taken from program 0412) may occur:

```
        CMP     0,12(SP)
        BNE     NONZER
        CLR     @2(SP)
        RTS     PC
NONZER:...
```

Due to a fault, the BNE instruction changes to a no-operation instruction. Consequently, instead of branching to NONZER and continuing the normal

path, it continues with the next two instructions and the program is terminated.

3) The program executes for an infinite time.

This is referred to as the infinite-loop case. These types of programs literally execute for an infinite time; they take more than an order of magnitude greater the time they would normally take to run. Programs that execute in less than an order of magnitude the time they would normally take are considered programs with wrong results. As an example, consider the following excerpt from program 0613:

```
BMT:    MOV     ROWPTR,-(SP)
         .
         .
         .
        RTS     PC
START:  MOV     STACK,SP
         .
         .
         .
NEXT:   MOV     (R0)+,-(SP)
         .
         .
         .
        JSR     PC,BMT
        DEC     R1
        BNE     NEXT
        HALT
```

Assume that DEC is changed to TST. Register R1 is not changed during the execution of subroutine BMT, since it is saved in the entrance to the subroutine and restored on returning. Also, it is not changed between the statement labeled NEXT and the JSR instruction. In the beginning it contains a value of one; this does not change. The TST R1 instruction sets the condition codes to nonzero. BNE NEXT always branches to NEXT and never ends.

4) The program traps to one of the PDP-11 trap locations.

Since RTI's program was run on a PDP-11 simulator without any operating system support and without any trap handler, trapping to a trap

location was treated as a termination point. Generally, this case changes to one of the three cases mentioned before if the execution continues after the trap or interrupt, even without any software support. An example case is one in which an instruction changes to an illegal, or unused instruction. Another example is an odd-address trap, as may occur in the 0613 program. There are several occurrences of the instruction CLR -(SP) in a subroutine. If this instruction changes to a no-operation instruction, the stack pointer has a wrong value. Returning from the subroutine, the program begins executing in the middle of the data. It executes one data word as MOVE X1(SP),X2(R2), and since X1 is an odd value, the final result of X1(SP) will be an odd address, which in turn causes an odd-address trap.

As discussed earlier in the measurement section (Section 3b(5)(c)), there were several parameters of interest to measure. However, several factors make those measurements invalid. So, no attempt was made to measure them, because they have changed the class in which a faulted result belonged. In other words, in one execution environment a fault might belong to the infinite-loop class, whereas for another execution environment the same fault might belong to the trap class. The factors which greatly determine to which of the four classes (manifestations at the system level) a fault belongs are:

1) The content of the memory cells which are not used by the program (i.e., the correct, nonfaulty program neither initializes nor uses these cells at all). Consider a program in which a branch instruction has changed to another branch, jump, or jump subroutine. Assume the destination (the PC after branching) is somewhere out of the program boundary. (It points to an unused portion of the memory). Depending on what is in that location and consecutive locations, different results might occur. If the location jumped to contains a zero, the program will halt. In other words, the program ends with a wrong result. If the location jumped to contains data from other programs, the program might halt or it might cause an illegal or unused instruction trap. If the location jumped to is at an instruction boundary in the middle of another program, the program might continue and end up with a wrong result, a trap or an infinite loop.

2) The contents of the architectural registers (the registers seen by the programmer) or implementation registers (the internal registers not known to the programmer) at the start of the program. In the following section of program 0613,

```
MOV     BASE(SP),ROWADR !ROWADR=%2
 .
 .
 .
TST     (ROWADR)+
```

assume that "MOV BASE(SP),ROWADR" is changed to a no-operation instruction. If the original value of ROWADR is an even number, the program may end with a wrong result. If the value is odd, it may trap to an odd address.

3) The location at which the program is loaded.

Some addresses used in the PDP-11 architecture are absolute addresses. If they are executed as instructions, they may cause different results. For example, if the ADD instruction in program 0613 is in location 000036 and the program is loaded from location 0000, then the content of locations 0 and 2 will be

LOCATION	VALUE
0000	062700
0002	000232

If a program is loaded from this location, the first word is changed to NOP (no-operation), and the second word is executed as an instruction. This, in turn, executes a RTS instruction, which can result in any of the four fault manifestations at the system level. If the program is loaded from location 1000,

LOCATION	VALUE
1000	062700
1002	001232

then the same fault causes 001232 to be executed as a BNE, which can cause a halt, a trap, or an infinite loop. If the program is loaded from location 5000,

LOCATION	VALUE
5000	062700
5002	005232

005232 is executed as an INC instruction and any of the four fault manifestations may occur.

(3) Data Storage Function Fault Injection

Data storage functions consist of all the registers used in a digital system. These registers can be architectural registers (registers seen and used by the assembler programmer) or implementation registers (internal registers of which the assembler programmer is not aware). The fault classes that have been considered for functional fault modeling are the classical stuck-at-zero or stuck-at-one faults. These faults manifest themselves as one or more of the bits in a register stuck-at-zero or stuck-at-one.

This section of the report concentrates on fault injection in the instruction register. Fault injection in the general-purpose registers is not discussed because the faults manifest themselves in the same manner as they do for the register-decode function [Section 4c(1)].

The ISP fault injection mechanism is used for fault injection in the instruction register. All 16 bits in the instruction register are stuck-at-zero and stuck-at-one. The results of the fault injection are presented in Appendix C(3). Remember that only single faults are injected, one at a time; multiple faults are avoided.

Since instructions are of variable length, as are the op-codes, fault injection in the instruction register implies the following. For a no-operand instruction, such as HALT, RTI, etc., a fault injected results in the execution of a wrong instruction. For a one-operand instruction, a fault injected in the ten most significant bits results in a wrong instruction; a fault injected in the six least significant bits results in

a change in the address mode and register, and generally, a change in the operand. For a two-operand instruction, fault injection in the four most significant bits results in a change of operation; a fault injected in the 12 least significant bits results in a change in address mode and register, and consequently, a change in the operand.

Four different manifestations were observed as the result of fault injection in the instruction register. They are discussed below according to the frequency of their occurrence, with the most frequent one discussed first. The remaining three occur with relatively the same frequency, which is far less than that of the first.

1) The program traps to one of the PDP-11/70 trap locations.

Two major types of trap are the odd-address trap and the red-stack trap which are stack and stack pointer related traps, respectively. Since no trap or interrupt vectors were loaded in the simulation runs and since there was no trap handler, this case was treated as a termination point. Had the program been continued beyond this point, an infinite loop or a wrong result would have occurred.

An example of this case is illustrated as part of program 0613 below.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	
1272	012706	001424	START :	MOV	#STACK,SP
1276	012700	001424		MOV	#TEST,R0
1302	012001			MOV	(R0)+,R1
1304	012046		NEXT :	MOV	(R0)+,-(SP)
1306	012046			MOV	(R0)+,-(SP)
1310	012046			MOV	(R0)+,-(SP)
1312	004767	177462		JSR	PC,BMT
1316	005301			DEC	R1

Assume that bit number 1 (the second least significant bit) of the instruction register is stuck-at-zero. The listing presents the instruction that would have actually executed had there been no fault, the instruction that executed instead in the presence of the fault, the PC (program counter), the SP (stack pointer), and the registers that were affected. Initially, all the general-purpose registers (including SP) were set to zero except PC, which was 1272 (the start location of the program).

ACTUAL INSTRUCTION	EXECUTED INSTRUCTION	PC	SP	RO	R1	R4
012706	012704	1276				1424
012700	012700	1302		1424		
012001	012001	1304		1426	0001	
012046	012044	1306		1430		1422
012046	012044	1310		1432		1420
012046	012044	1312		1434		1416
004767	004765	1316	177776			

As it is indicated above, the fault injected causes R4 (register four) to be operated upon instead of SP (register six) in the first six instructions. In the seventh instruction, which is a JSR, SP is used for saving the return address. However, since SP has an initial value of zero, when it is decremented, it causes a stack overflow and, consequently, a red-stack trap. Note that the outcome could have easily changed to an odd-address trap, if the initial value of SP were an odd number.

2) The program ends with a wrong result.

Consider the program 0412. When bit 0 (the least significant bit) is stuck-at-zero,

LOCATION	VALUE	VALUE	VALUE	LABEL	INSTRUCTION	
1000	012706	001446		START :	MOV	#STACK,SP
1004	012746	001212			MOV	#STR,-(SP)
1010	016746	000174			MOV	STRLEN,-(SP)
1014	012746	001242			MOV	#ARG,-(SP)
1020	016746	000214			MOV	ARGLEN,-(SP)
1024	012746	001206			MOV	#LOC,-(SP)
1030	004767	000002			JSR	PC,CHRSRC
1034	000000				HALT	
1036	022766	000000	000012	CHRSRC:	CMP	#0,12(SP)

and the program is executed as indicated.

ACTUAL INSTRUCTION	EXECUTED INSTRUCTION	PC	SP	CONTENT OF LOCATION WHERE SP IS POINTING TO
012706	012706	1004	1446	0000
012746	012746	1010	1444	1212
016746	016746	1014	1442	0026
012746	012746	1020	1440	1242
016746	016746	1024	1436	0003
012746	012746	1030	1434	1206
004767	004766	1034	1432	1034
		1436		
000003	000002	1440	1434	
		1034	1436	
000000	000000	1036		

Notice that the program execution continues in a normal fashion up to location 1030, which has the "JSR PC.CHRSRC" (004767). At this point and as a result of the fault injected, "JSR PC,CHRSRC" (004767) changes to "JSR PC,X(SP)" (004766). The value of X (the index) is stored in the location where PC (1032) is currently pointing (2). The destination address is then calculated by adding the current value of SP (1434) to the value of X (2), or 1434 + 2 = 1436. The current PC (1034) is saved on the stack and the destination address (1436) is loaded into the PC: the program then continues execution from location 1436. In location 1436, SP has a value of 000003 which is a BPT instruction. However, because the fault is injected, it executes the program as an RTI (000002) instruction, and control returns back to the instruction after JSR. The next instruction executed is a HALT instruction (location 1034).

3) The program is in an infinite loop.

As an example of an infinite loop manifestation, consider program 0613 when bit number 11 (the 12th least significant bit) of the instruction register is stuck-at-zero. Part of the program is as follows.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	
1272	012706	001424	START :	MOV	#STACK,SP
1276	012700	001424		MOV	#TEST,R0
1302	012001			MOV	(R0)+,R1
1304	012046		NEXT :	MOV	(R0)+,-(SP)
1306	012046			MOV	(R0)+,-(SP)
1310	012046			MOV	(R0)+,-(SP)
1312	004767	177462		JSR	PC,BMT
1316	005301			DEC	R1

Execution of the program is as follows.

ACTUAL INSTRUCTION	EXECUTED INSTRUCTION	PC	SP	R0	R1
012706	012706	1276	1424		
012700	012700	1302		1424	
012001	012001	1304		1426	0001
012046	012046	1306	1422	1430	
012046	012046	1310	1420	1432	
012046	012046	1312	1416	1434	
004767	000767	1272			

As the listing shows, all instructions are executed correctly to the point where "JSR PC,BMT" (4767) is fetched. At this point and as a result of the fault, "JSR PC,BMT" (004767) changes to "BR X" (767). Using the offset part of the branch instruction (the 8 least significant bits of the instruction, 367), the new value of PC will be 1314+2*(-22)= 1272, which is the starting address of the program. This sequence continues as an infinite operation.

4) The PDP-11/70 WAIT instruction is executed.

The only reason this case was included as a separate case is to demonstrate ISP limitations. In the current version of the PDP-11/70 ISP description, an ISP wait statement is executed whenever a PDP-11/70 WAIT instruction is desired. However, since there is nothing for the ISP to wait on, it will issue an error message and stop. In the actual PDP-11/70, a WAIT instruction execution causes the CPU to relinquish the bus control until an external event occurs. Then it continues execution. Regarding the actual PDP-11/70 WAIT instruction execution, this manifestation will be converted to one of the other three manifestations.

An example of this case is the program 0412 execution when the bit number zero (the least significant bit) of the instruction register is stuck-at-one.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	
1000	012706	001446	START :	MOV	#STACK,SP
1004	012746	001212		MOV	#STR,-(SP)

The first instruction changes from 012706 to 012707 (MOV #STACK,PC). As a result, the PC is loaded with 1446 (the address of the variable STACK). Location 1446 is in the program boundary and has the value 000000. When the program tries to execute the 000000 instruction, the instruction changes to 000001 as a result of the fault. The 000001 instruction is a WAIT instruction.

In summation, there are several results worth mentioning in data storage functional fault injection.

1) Most of the manifestations were the PDP-11/70 trap, especially the odd-address trap and the red-stack trap.

2) It was noted earlier that, if continued, the PDP-11/70 trap changes to a wrong result or an infinite loop. This is not true for the instruction register when one of its bits is stuck-at-one. No HALT instruction can be executed because the HALT instruction op-code is 000000 and with a faulted instruction register, at least one bit is on at all times. As a result, the PDP-11/70 trap, if continued, will only change to an infinite loop.

3) The same factors affecting the instruction decode function outcome will affect the data storage function outcome.

4) Fault manifestations are not dependent on the location of the bit faulted in the instruction register.

(4) <u>Data Manipulation</u>

As stated earlier, the PDP-11/70 uses a TI 74181 ALU chip. Of the 48 functions available, 30 are used in the PDP-11/70. In this section, functional fault modeling is applied to the data manipulation function of the PDP-11/70 ALU.

Section 3b(5)(c)4 states that the three major functional faults (gate-level faults manifested at the chip level) are:

1) A constant operation in output. In other words, the output of the ALU is exclusively ORed with a constant m, or the output is added/ subtracted with a constant n.

2) A change of operation. Instead of adding two numbers, they are exclusively ORed, or they are subtracted.

3) An output bit stuck-at-zero or stuck-at-one. Note that this case does not include the two previous cases.

In order to inject these three types of faults, the PDP-11/70 ISP description required modification. The PDP-11/70 ALU is described as a module with the control lines as the input to the module and the result as the output of the module. For a change of operation the ISP fault injection was used to inject faults in the control lines. Two parameters were added for the purpose of faulting the output with an exclusive OR or add/ subtract operations. These were FAULT<1:0> and MASK<15:0>. FAULT specifies whether the output of the ALU should be faulted and, if so, what operation should be performed. Table 14 shows the possible values of FAULT and the meaning of each.

Table 14. Possible Values of Variable FAULT and the Action Taken.

FAULT	Meaning
0	No fault
1	Final output = correct output $\oplus$ "MASK"
2	Final output = correct output + "MASK"
3	Final output = correct output - "MASK"

MASK is a 16-bit register which combines with the correct result of the ALU to produce the faulted result.

The following method was used for all data manipulation fault injection. Faults were inserted before the program started and were present throughout the program execution. During the fault injection process, only one fault was injected at a time; multiple faults were avoided.

Appendix C(4) lists all the data manipulation functional faults which were injected in the ALU and their manifestations while the two CFA programs mentioned earlier (Section 4b(1)) were executing. There are four resulting cases. These are:

1) The program being executed traps to one of the PDP-11 trap locations.

Trap was considered as a termination point. This is due to the fact that the programs being injected with faults were running in a PDP-11/70 simulator without the loading interrupt and trap vectors. Consequently, there were no interrupt and trap handlers nor any operating system support. Generally, this case would have changed to one of the other resulting classes (a wrong result or an infinite loop). Stack and stack pointer related traps and interrupts, such as stack overflow, were the most frequent traps and interrupts that occurred.

An example of a PDP-11 is the program 0613 when the output of the ALU is exclusively ORed with 4. Part of the program is listed here for more detailed study.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	OPERANDS
1272	012706	001424	START:	MOV	#STACK,SP
1276	012700	001424		MOV	#TEST,R0
1302	012001			MOV	(R0)+,R1
1304	012046		NEXT :	MOV	(R0)+,-(SP)
1306	012046			MOV	(R0)+,-(SP)
1310	012046			MOV	(R0)+,-(SP)
1312	004767	177462		JSR	PC,BMT
1316	005301			DEC	R1

Two things should be kept in mind to understand exactly what is happening. First, all the operations are performed through the ALU. For example, to increment the PC by 2, the PC is passed through the ALU, or to load any of the general-purpose registers from the memory data register, the data is passed through the ALU. Secondly, to fetch an instruction, the PC passed through the ALU and is loaded into a register called VAR (virtual address register). The PC again passes through the ALU to be incremented, to point to the next instruction or to the data portion of the current

instruction. Keeping all of this in mind, the sequence of events affecting the registers are as follows:

Initial value of PC = 1272

	VAR	PC	SP	RO	R1	INSTRUCTION FETCHED
1	1276	1270				012700
2	1274	1276		1420		
3	1272	1304				012706
4	1300	1302	1420			
5	1306	1300	1412	1426		012046
6	1304	1306	1414	1434		012046
7	1302	1314		1432	1454	012001
8	1310	1312	1416	1430		012046
9	1316	1310			1443	005301
10	1314	1316				177462

In row 1, PC (1272) goes through the ALU to be loaded into the VAR and becomes 1276. On the second pass through the ALU, PC is incremented by 2 and becomes $(((1272 + 2) \oplus 4) = 1270)$. Using the VAR, the first instruction is fetched from location 1276, which is "MOV #TEST,RO." In row 2 the same sequence is followed for fetching the second part of the instruction. Because of the fault injected, it is fetched from location 1274, which, by chance, has the same value as the correct instruction. The number fetched is 1424, but when passed through the ALU, it changes to 1420 and is loaded into register zero. This sequence continues and instructions and data are fetched up until the tenth row. At this point, the address of the instruction to be fetched is 1314 (content of VAR), which is pointing to the second word of a two-word instruction. The value of location 1314 is 177462, which represents a floating point instruction. Since floating point instructions are not implemented in the current PDP-11/70 ISP description, an illegal instruction trap occurs.

2) The program ends with a wrong result.

A wrong result here means that the program terminates at some point. The termination point can either be the point where the program would normally stop or any other point. This obviously happens when the machine tries to execute instruction 000000, which is a halt instruction. An example of the halt-instruction case is program 0412 which follows.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	OPERANDS
1000	012706	001446	START :	MOV	#STACK,SP
1004	012746	001212		MOV	#STR,-(SP)
1010	016746	000174		MOV	STRLEN,-(SP)
1014	012746	001242		MOV	#ARG,-(SP)

The fault injected in the ALU causes the correct output of the ALU to be incremented by two. The program proceeds as follows.

Initial PC value = 1000

	VAR	PC
1	1002	1004
2	1006	1010

In the row 1 the PC is incremented to 1004 and the instruction fetched is from location 1002. (Normally, these locations are 1002 and 1000, respectively.) The fetched instruction (001446) is the second word of the actual instruction (012706, 001446). The fetched instruction (001446) is a BEQ instruction in which the condition for branching is not satisfied, due to the initial state of the system. The PC is now incremented by four to become 1010. The instruction to be fetched is from location 1006 which again is the second word of a two-word instruction (001212). This two-word instruction is a BNE instruction. This time, the condition for branching is satisfied and the offset of the branch instruction causes PC to be updated to 436. Since the program starts from location 1000 and since no other user is running and no trap or interrupt vectors are loaded, locations 436, 440 contain zero and a halt occurs.

3) The program is in an infinite loop.

The same concept applied in the instruction decode case applies here. An example of the program 0613 follows.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	OPERANDS
1270	100000			.WORD	100000
1272	012706	001424	START :	MOV	#STACK,SP
1274	012700	001424		MOV	#TESTS,R0

Assume that the fault injected causes the final result of the ALU to be the actual result (result for a nonfaulty ALU) exculsively ORed with two.

Initially, PC is located in 1272. When PC passes through the ALU to be loaded in the VAR, it changes to 1270. Consequently, the instruction is fetched from location 1270, which is 100000. In the second pass through the ALU and before instruction fetch, the PC changes to 1276 $((1272 + 2) \oplus 2)$. The instruction being fetched (100000) is a BPL instruction with 0 as the offset for branching, which forces PC to 1274 $((1276 + 2 * 0) \oplus 2)$. From this point the instruction in location 1276 is being fetched and PC remains at 1274 $((1274 +2) \oplus 2)$. Consequently, the program execution is in an infinite loop.

In conclusion, several points are worth noting in the data manipulation fault injection:

1) All the faults injected manifested themselves very quickly -- on the average, in the first five or six instructions executed. The reason was that almost every operation performed uses the ALU.

2) The same factors affecting the instruction decode function outcome affected the ALU data manipulation outcome.

3) Most of the manifestations observed here were PDP-11/70 traps. This was apparently due to the rapid change in PC and SP values as a result of the ALU faults. As stated earlier, the PDP-11/70 trap was viewed as a termination point because there was no operating system support, no interrupt or trap vector was loaded, and there was no interrupt handler.

(5) Results of the Overall Fault Injection

In Sections 4c(1) - 4c(4) faults were injected into the four functional modules of a single CPU. The following observations can be made concerning all four functional modules.

1) Injected faults manifest themselves in four different ways:

   a) The program ends with a wrong result. The program can terminate at the point where it would normally stop or it can terminate at any random point.

   b) The program causes a PDP-11/70 trap. This is usually caused by executing a nonexistent instruction, a word instruction with an odd address, or a stack overflow.

c) The program execution lasts much longer than it would normally, or the program execution is infinite . When the program takes much longer than it normally should, different parts of the program are executed at random. When program execution is infinite, the program branches over a small group of instructions.

d) The program ends with a right result. The faulted part of the program does not affect the program execution (e.g., an instruction which is faulted but never executed), a branch condition is never satisfied, or a register is faulted but never used.

2) Several factors affect the fault manifestations. They are:

a) The content of memory cells which are not occupied by the program. Execution of the program on a nonfaulty system will never initialize or use these memory cells.

b) The content of architectural registers and internal registers.

c) The location where the program is loaded; i.e., the address of the locations where the program and the corresponding data reside.

3) The PDP-11 architecture trap mechanism is an effective tool for detecting faults which manifest themselves as traps. It would be very helpful if there were a simple way to distinquish between user-introduced errors that cause the trap and hardware-failure traps.

4) Faults manifest themselves relatively quickly after they become effective; i.e., whenever they begin to cause a wrong result.

d. <u>Dual-CPU Fault Injection</u>

The dual-CPU configuration introduced in Section 2b was originally designed as an aid to measure fault latency (the time from the original occurrence of a fault in one CPU until the time a difference between the memory busses of both CPUs occurs). As discovered in this study, fault latency is very dependent on both the environment and the state of the machine. (See Section 4c) Because of these two factors, RTI decided not to measure fault latency.

Instead, the dual configuration was used as a BIT, albeit a very high-level BIT. In this instance, a fault was injected into one CPU and the simulation was halted whenever the memory busses of both CPUs differed.

If the simulation stopped, this was an indication the BIT was working. Whether the BIT was effective or not is another matter. If the simulation ran into a system-level fault manifestation, then the BIT was not detecting the fault. The same fault classes are discussed in this application as with the single CPU.

(1) Register Decode

Faults injected in the dual CPU were the same as those injected in the single CPU, thus allowing comparison of the results between these two applications. These three classes of faults are discussed below as they apply to the dual CPU.

1) The wrong register is selected.

As seen in Table 15, all the faults were detected by the BIT. They were detected as soon as, or sooner than, the same faults in the single-CPU configuration. Comparing fault XXO in Table 10 (single CPU) with that in Table 15, note that, for the single CPU, data was executed and for the dual CPU, the BIT stopped on an address disagreement. The address which caused the disagreement is the address of the data that was executed by the single CPU. The disagreement occurred in the instruction fetch cycle.

2) No register is selected.

Table 16 tabulates RTI's results for this category. Again, all the faults were detected by the BIT. For this category the improvement in fault latency was very good. A quick comparison with the single CPU in Table 11 indicates that the single CPU program ran to completion in several cases but produced wrong results, while for the dual CPU, the fault was detected and no erroneous results were produced. The most significant points to be made here are that 1) the fault latency is drastically reduced in many cases and 2) that all the faults were detected, which was not always the case in the single-processor configuration.

3) Multiple registers are selected.

This category is tabulated in Tables 17 and 18, and the results are very similar to those for the previous category. All the faults were again detected by the BIT, by noting differences in the address lines. These addresses were all pointing out of the instruction bounds.

Table 15. Wrong Register Is Selected.

Register File Index is Stuck-At	Results	Comments
XX0	BIT detected	Address of stack put in R7.
X0X	" "	Address disagreement at checker.
0XX	" "	" " " "
X00	" "	" " " "
XX1	BIT detected	Address of stack put in R7.
X1X	" "	Data disagreement at checker.
1XX	" "	" " " "
X11	" "	Address of stack put in R7.

X = do not care

Table 16. Register Is Not Selected.

Register Not Selected	Results	Comments
R0	BIT detected	MOV (R0)+,R1; address disagreement.
R1	" "	MOV R1,-(SP); data disagreement.
R2	" "	
R3	" "	
R4	" "	
R5	" "	
R6	" "	
R7	" "	

Table 17. Desired Register Is Included.

TECHNOLOGY	Register Selected	Results	Comments
OR			
	1,3,5	BIT detected	*PC modified;address difference
	2,4,6	"	" " " "
	5,7	"	*PC forced to value of SP; address difference.
	6,7	"	"" "" " "
AND			
	1,3,5	BIT detected	*PC modified forced to zero; pointed at system data area. This occurred because the contents of the registers were all set to zero when the program was loaded. Address difference detected.
	2,4,6	"	
	5,7	"	
		"	

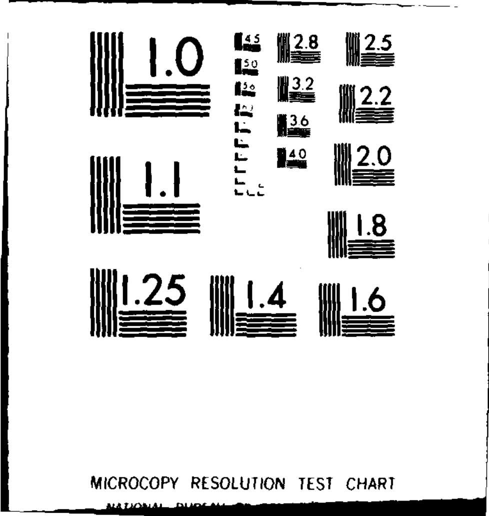
Table 18. Desired Register Is Excluded.

TECHNOLOGY	Register Selected	Results	Comments
OR			
	1,3,5	BIT detected	*Contents of the registers were all zero. Forced the PC to zero. Address difference detected.
	2,4,6	"	
	5,7	"	PC was forced to the value of the SP as in part (a) of this table. Address difference detected.
	6,7	"	
AND			
	1,3,5	data executed	*Registers were all zeros. PC was forced to zero. Address difference detected.
	2,4,6	"	
	5,7	"	" "
	6,7	"	" "

AD-A093 735 RESEARCH TRIANGLE INST RESEARCH TRIANGLE PARK NC SYST--ETC F/G 9/2
DEVELOPMENT OF A METHODOLOGY FOR VERIFYING MILITARY COMPUTER FA--ETC(U)
SEP 80 J B CLARY, R K JOOBBANI, F M SMITH DAAK80-79-C-0780
UNCLASSIFIED RTI/1822/00-01F CORADCOM-80-0780-F NL

2 of 2

END
DATE
FILMED
2-81
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART

In conclusion, the BIT appeared to work quite well for these classes of functional faults. It was especially useful in the no-register-selected category, where previously many faults went undetected. The BIT stopped data corruption in the memory, but it did not stop the corruption of the registers. This was due to the BIT itself, where it did its checking. The BIT would be a more useful mechanism if it returned an error code indicating which set of lines on the bus (data lines, address lines, control lines) caused the disagreement. This would allow the recovery process to determine what should be done to recover from the fault.

(2) Instruction Decode

Fault injection in the instruction decode function in the dual CPU is the same as that explained for the single CPU. The program is stopped whenever a discrepancy between the memory busses of the CPUs occurs.

Using the dual CPU as a system with BIT, four classes of outcomes were distinguished. They are presented below, along with examples. The frequency of occurrence for the first outcome was much greater than for the other three.

1) The dual CPU detects the faults.

The dual CPU was very effective in catching most faults -- on the average, after five or six instruction executions from the point of fault injection.

An example of this case is taken from program 0613. Assume that for one of the CPUs, the JSR instruction has changed to an NOP (No-operation) instruction. In other words, the faulty CPU (CPUA) tries to execute the following line of code:

LOCATION	VALUE	INSTRUCTION
1312	000240	NOP
1314	177462	NON-EXISTENT INSTRUCTION

whereas the correct CPU (CPUB) tries to execute the following:

LOCATION	VALUE	VALUE	INSTRUCTION	
1312	004767	177462	JSR	PC,BMT

Now, both CPUs have fetched the content of location 1312. One sees the content as an NOP. (Notice that the content of location 1312 in memory is not changed, but it is changed in CPUA after the fetch cycle.) The other sees it as JSR. Both CPUs return to fetch the content (177462) of location 1314, CPUA fetching 1314 as an instruction, CPUB as an operand. Since they are both fetching the content of the same location, no discrepancy is detected, even though the two CPUs are in different phases. After fetching 177462, CPUA decodes the content as an instruction; since it is a floating point instruction and it is not implemented, a nonexistent instruction trap occurs. CPUA saves the content of PC and PSW in a temporary register and puts the trap vector address (10) into its own virtual address register (VAR) to fetch the new PC and PSW. In the meantime, CPUB wants to push the return address (1316, current PC value) into the stack, so it decrements the SP and puts it (1414) into its own virtual address register. The two addresses are now different and the *dual CPU checker raises an error flag.*

2) The dual CPU does not detect the fault and the result is correct.

In this case, the faulted instruction is not executed or the condition for a faulted branch instruction is never satisfied. Consider the part of program 0412 listed below:

```
CHRSRC:    CMP    #0,12(SP)
           BNE    NONZER
           CLR    @2(SP)
           RTS    PC
NONZER:    MOV    R0,-(SP)
```

Assuming that CLR is faulted, "BNE   NONZER" is satisfied, the program control is transferred to the NONZER location, and "CLR   @2(SP)" is never executed. Remember that this is the only CLR instruction in the program.

3) The dual CPU does not detect the fault and program execution stops.

This case is mostly one of the ISP symptoms. In other words, if one of the CPUs executes a HALT instruction which is directly translated into an ISP stop statement, the whole simulation stops.

An example of this case is the program 0412 execution. Assume that the injected fault causes all the CMP instructions to change to NOP instructions. A line of the program is listed below for more detailed study:

LOCATION	VALUE	VALUE	VALUE	LABEL	INSTRUCTION	
1036	022766	000000	000012	CHRSRC:	CMP	#0,23(SP)

Both CPUs fetch the instruction at location 1036 (022766). The faulty CPU (CPUA) executes the instruction as an NOP, whereas the nonfaulty CPU (CPUB) executes the instruction as a CMP. In the next cycle, CPUA fetches the content of location 1036 (000000) as an instruction, which is a HALT instruction. CPUB fetches the content as an operand. No error is detected in either case because both of the CPUs are pointing to the same location, although for different reasons. After CPUA executes the HALT instruction, the entire simulation stops.

4) The dual CPU detects a fault which does not exist (a false alarm).

This is usually the result of unidentical system states at the start of a simulation -- states which are largely due to the way the CFA programs were written. Both of the test programs save the contents of the general-purpose registers and, on return, they restore them. For example, R0 in one CPU may have a zero in it, whereas R0 in the other may have a four. At the start of the subroutine, these two registers will be saved somewhere in memory (one location). In the process of saving them, the checker compares the two values. Since they are different, the checker raises a flag, even though the values might not be used at all.

In summation, several conclusions were obtained from using the dual CPU as a system with BIT.

1) Faults are usually detected and they are detected relatively quickly. Fault latency can be improved by adding some type of mechanism which not only checks the memory busses of the two CPUs, but also checks the phases of the two CPUs. In a previous example the two CPUs were fetching the same location (memory) word from the CPU but for different purposes, one as an instruction and the other as an operand. If an indicator had shown whether the two CPUs were in the instruction fetch cycle, the operand fetch cycle, or execution cycle, this error would have been detected sooner.

2) Correct operation of the checker and the rate of false alarms depend on the state of the system at the beginning of program execution and after each fault is detected. In other words, the two CPUs should be in an identical state at the beginning of program execution.

3) Fault latency is strongly dependent on the programmimg style and the algorithm used. In the dual CPU only memory accesses are checked. If the contents of internal registers differ because of the fault injected, it will not be detected until the registers are used as a memory address or they are being stored in memory. For example, assume that "DEC R1" is executed in one CPU as an NOP and as "DEC R1" in the second. The two R1s are different. If the register is not saved or used immediately as a pointer to a memory location, the fault will not be detected until R1 is used later as an address or as data to be stored in memory.

(3) Data Storage Function Fault Injection in Dual CPU

Fault injection in the data storage module is handled in the same manner in the dual CPU as in the single CPU. A single fault is injected, only the instruction register is injected with a fault, and the fault exists throughout the program execution. Since the instruction register is used once in every instruction cycle, the fault manifests itself very quickly and is in turn detected very quickly.

On the average, the faults manifested themselves in the first five or six instructions. The dual CPU caught all of the faults injected in the next one or two instructions. The longest time for a fault to become effective was after 123 instructions that occurred when bit 15 of the instruction register was stuck-at-zero.

(4) Data Manipulation Function Fault Injection in Dual CPU

Again, fault injection in the data manipulation function in the dual CPU is the same as in the single CPU. The discussion here concentrates on the outcome of fault injection in the dual CPU. In general, there were three outcomes.

1) The fault is detected.

In this case, the dual CPU detected the fault and terminated program execution. This was the most frequent occurrence. The fault was detected very quickly, due to the fact that the ALU was used for every calculation.

As an example of this case, look at the execution of program 0613 when the final result of the ALU was exculsively ORed with 2. At the beginning of program execution both PCs are set to 1272. To fetch the instruction at location 1272, both PCs pass through their respective ALUs. The outcome of the faulty ALU is 1270 ($1272 \oplus 2$), whereas the outcome of the correct ALU is 1272. Since these two numbers are compared and they do not agree, an error flag is raised and the program execution stops.

2) The fault is not detected and the program stops.

This is primarily due to the way the HALT instruction is executed. As mentioned earlier, a HALT instruction causes an ISP stop statement to be executed. This case can be easily changed to the previous outcome if there is a time-out mechanism.

Consider the execution of program 0613 when an ALU fault causes all the logical operations (M=L, see Appendix B(1)) to change to arithmetic operations (M=H). At the beginning of program execution, the PCs in both CPUs are initialized at 1272. The PCs are passed through the ALU to be loaded into each corresponding VAR to fetch the instruction. Function B of the TI 74181 is used for a straight PC move through the ALU. In the faulty ALU this function changes to (A+B)PLUS AB PLUS 1 and causes the VAR of the faulty CPU to be loaded with a very large number, which is pointing to the I/O page. Since the I/O page accesses are not checked by the checker, the faulty CPU fetches a word from the I/O page which has zero as its value. In the meantime, the bus checker is waiting for the faulty CPU to load its VAR and set the go flag. After fetching the zero from the I/O page, the faulty CPU executes it and causes the simulation to terminate. The addresses referencing the I/O page are not checked by the checker because the I/O page has the general-purpose registers and other registers directly accessible by the instructions; as such, we consider them part of the CPU.

3) Program executes infinitely.

Such a case occurs when the final result of the ALU is always an odd number, or when the final result of the ALU is exclusively ORed or

added/subtracted by an odd number. When the PC is passed through the ALU, it has an odd value; as a result, an odd-address interrupt occurs. The ALU is used to handle the interrupt and since its output is always odd, it in turn causes another odd-address interrupt. This condition continues indefinitely, while the bus checker continues to wait for the faulty CPU to load its VAR and set the go flag.

To summarize, some of the results of data manipulation fault injection are listed below:

1) Three types of outcomes occurred from injecting faults in the data manipulation module of the dual CPU.

2) The dual CPU was able to catch most of the faults as soon as they manifested themselves.

3) Although the last two outcomes occurred as a result of the ISP description, they can easily occur in an actual system.

4) If a method such as time-out is used, the last two outcomes can be changed to the first outcome.

(5) Results of the Overall Fault Injection in the Dual CPU

In general, the dual CPU BIT works relatively well in detecting and catching most functional faults, despite the fact that it requires twice the hardware and that both CPUs must be in the same initial state. Several improvements can be made to increase BIT performance. These are:

1) Errors can be detected faster if the addresses, the data to and from the memory, and the corresponding CPU phases are checked. In every memory reference check, the phases of the two CPUs should also be checked to see if they are the same and they are referencing memory for the same purposes.

2) Fault latency can be reduced sharply if all the I/O page accesses (including general-purpose register) are checked. In the current simulation the I/O page accesses were excluded from the checker.

3) A time-out mechanism can prevent one CPU from waiting for an indefinite period of time for the second CPU. Such a case occurs when one CPU halts or passes into an infinite loop.

5. REFINED FUNCTIONAL FAULT MAPPING

a. Introduction

Early in the discussion on functional fault modeling, it was noted that gate-level fault manifestations at the chip (pin) level or functional level might be sharply reduced compared to the number of faults at the gate level. This is true if we are only considering the number of different manifestations. If one is concerned with fault injection at the functional level, there is not much difference between the number of faults inserted at the gate level and the number of faults inserted at the functional level. It is possible that functional fault injection costs more than gate-level fault injection.

To understand fault mapping from one space to another space, Figure 13 has been redrawn as Figure 20. Comparison of Figure 20 with Figure 13 shows that mapping of gate-level faults to chip-level faults is not a one-to-one occurrence. In other words, one gate-level fault might map into several chip-level manifestations. To clarify this point consider an ALU that can perform six different functions (A, B, C, D, E, F) and assume that the possible number of gate-level faults is eight (fault number 1, 2, ..., 8). Figure 21 is drawn with respect to the functions performed by the example ALU. Note that a gate-level fault might map into different manifestation classes, depending on the function. Also, for dependent and irregular fault classes, one gate-level fault might map into different manifestation classes, even within the same function. In addition, note that one manifestation class (labeled M_0, M_1, M_2, ...) is actually a no-fault manifestation class (M_0).

A no-fault manifestation class, hence referred to as a no-manifestation class, can be better understood if the following example is examined. Refer to the OR gate shown in Figure 22 with inputs A, B, C and output D. If the OR gate is stuck-at-one when all the inputs are zero, the stuck-at-one manifests itself at the output D. When the gate is stuck-at-one and at least one of the inputs is one, the stuck-at-one does not manifest itself at the output because D should be one. This is a no-manifestation class.

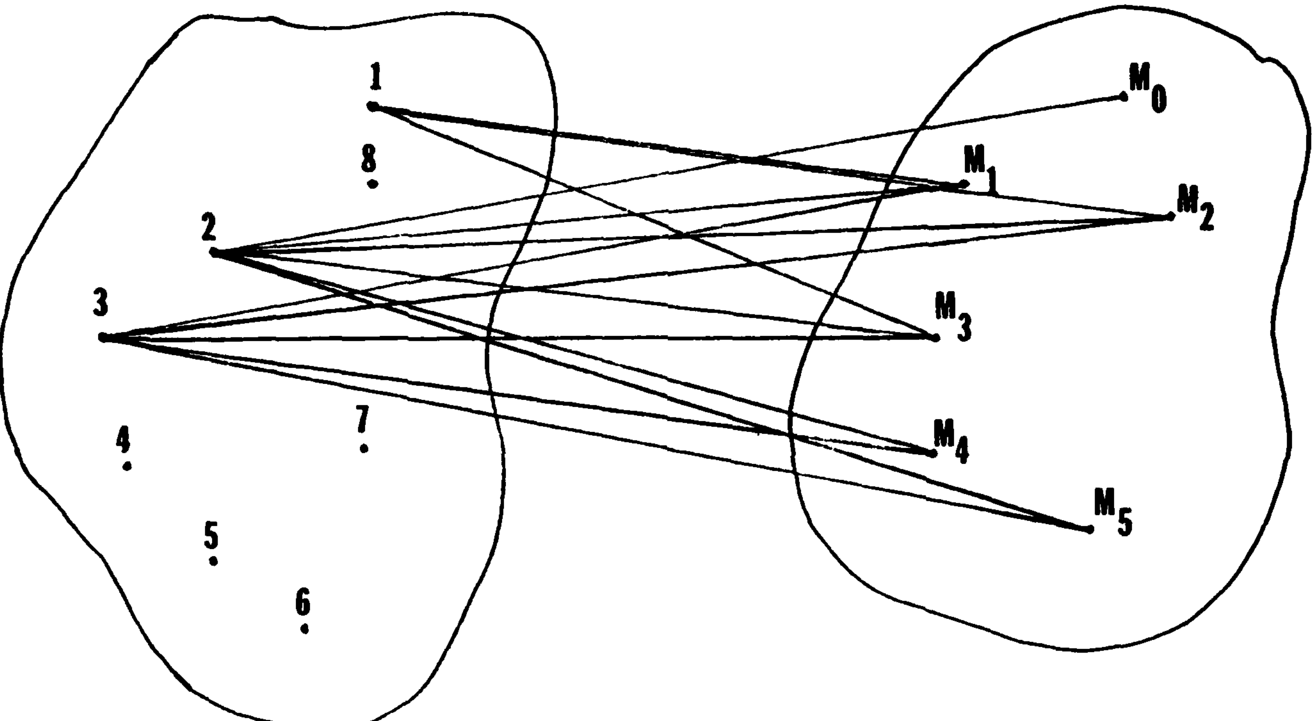


Figure 20. Digital System Fault Propagation from Gate Level to the Functional Level.

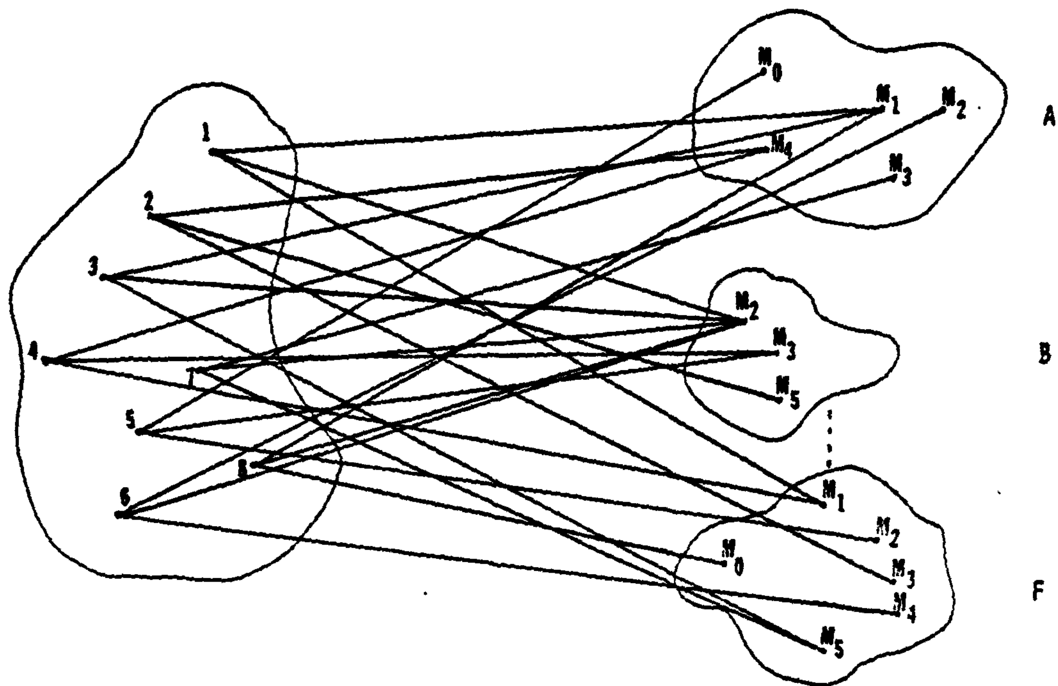


Figure 21. Digital Systems Fault Propagation from Gate Level to the Functional Level for Each Function.

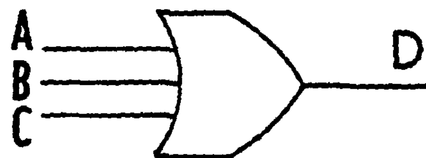


Figure 22. An OR Gate with Three Inputs.

Notice that the union of manifestation classes for all the functions is the universal set of internal faults manifested at the chip (pin) level.

b. Functional Fault Simulation Versus Manifestation Simulation

Functional fault simulation is useful for two reasons. First, the final result of gate-level fault manifestations at the functional level is very small compared with the set of gate-level faults. The result is useful if one is only concerned with manifestations, or if one is concerned with undesired event handling, different gate-level fault manifestations at the functional level, and the system or user level.

Second, fewer fault classes are required for fault injection at the functional level. Unfortunately, this does not reduce the effort involved for fault injection, and it may even require more effort.

For example, consider the example ALU mentioned earlier and assume that all faults injected are independent of input and are regular. If the M_1 manifestation is simulated (see Table 19), several gate-level faults are occurring at one time. That is, when function A is executed, faults 1, 3 and 8 are injected; when function C is executed, faults 1 and 8 are executed, and so forth. If function B is executed and the manifestation M_1 is still injected, then multiple gate-level faults are inserted to get manifestation M_1, which does not appear in the row corresponding to function B. Remember that faults 1 through 8 are single stuck-at faults. The number of faults to be injected collapses if two columns are exactly the same; in this example, columns 1 and 8.

Table 19. Pin-Level Fault Manifestation for a Hypothetical ALU.

Function \ Fault	1	2	3	4	5	6	7	8
A	M_1	M_4	M_1	M_4	M_0	M_2	M_3	M_1
B	M_2	M_5	M_2	M_3	M_3	M_2	M_2	M_2
C	M_1	M_0	M_3	M_2	M_4	M_2	M_0	M_1
D	M_3	M_2	M_4	M_1	M_4	M_5	M_4	M_3
E	M_1	M_1	M_4	M_5	M_3	M_5	M_0	M_1
F	M_1	M_3	M_5	M_0	M_2	M_4	M_5	M_1

It should be noted that this pertains to the internal chip failures and not to the external chip failures, such as pin faults. The difference between external and internal faults is that external faults can be present at all times. They are independent of inputs and control functions to the chips, whereas internal fault manifestations at the chip level depend on inputs and control functions. Simulating fault manifestations for every function and for every input to the chip is not really an internal fault simulation, but an external fault simulation. Internal faults manifest themselves as pin faults at the chip level and not every pin fault is a result of a single internal fault. Internal fault manifestations are a subset of pin faults, and there is a high correlation between internal faults and pin faults.

Part of Table 19 (for function A) is shown in Table 20 for the case in which some of the fault manifestations depend on the input to the ALU and are regular faults.

Table 20. Fault Manifestation for Dependent and Regular Faults.

Function \ Fault	1	2	3	4	5	6	7	8
A	M_0	M_0	M_1	M_4	M_1	M_0	M_3	M_1
	M_1	M_4				M_2		

When fault 1 exists for some combination of inputs, manifestation M_1 occurs, and for the remainder of the inputs, manifestation M_0 occurs, which is a no-manifestation fault class. If M_1 is injected when function A is being executed, regardless of inputs to the ALU, in reality, multiple fault injection is being simulated.

For the case of independent and irregular or dependent and irregular faults, the number of entries in each of the row and column intersections of Tables 19 and 20 will increase. For a 4-bit slice of the TI 74181 ALU, the maximum number of manifestations in one column and row intersection will be 2^8 = 256. Since there are 62 gates and two types of faults (stuck-at-zero and stuck-at-one), there are 62*2 faults. Using the 30 functions used by the PDP-11/70, the fault breakdown in Figure 23 occurs.

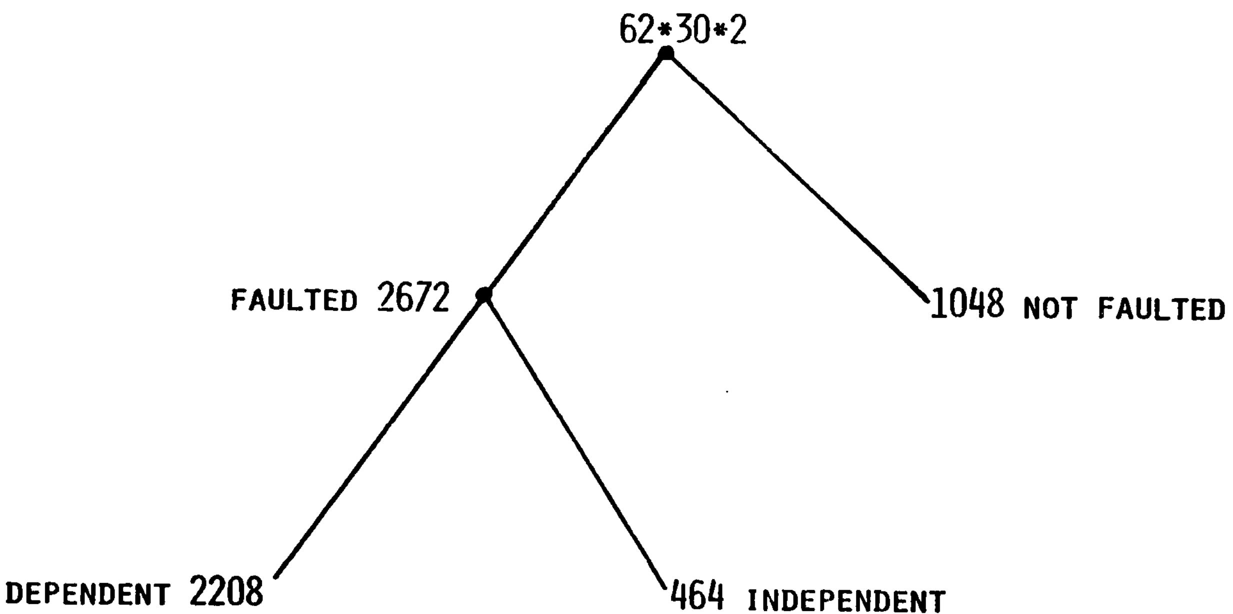


Figure 23. Breakdown of Number of Faults in the 74181 ALU.

Note that about one-fifth of the faults are independent; the rest are dependent. Assuming that all of the faults are regular, one can see that there is still a large number of cases to be considered.

The number of faults to be injected are reduced if one is careful of the way manifestation injection is used. That is, when a manifestation is injected, one makes sure that only the functions which cause that manifestation to occur are executed, not the others. Refer again to Table 19. If manifestation M_1 is simulated, only functions A, C, D, E and F are allowed to be executed, not function B. Thus, fault injection is sharply reduced. This method can be extended to the case in which specific functions and specific inputs are to be considered.

c. More Detailed ALU Fault Injection

Generally, a chip fault is caused by pin faults and die faults. Pin faults are usually independent of the input to the chip; they exist all the time. The functional fault injection explained in the previous section was a pin-fault injection. Die faults that contribute to gate-level faults are dependent upon the input to the chip. However, the only communication with the outside world is through the pins; gate faults manifest themselves as pin faults. This manifestation looks like a pattern-sensitive pin fault.

In order to predict correctly gate-level fault manifestations at the functional and system level, the ALU was described at the gate level in ISP. In other words, a 4-bit slice of the TI 74181 ALU was described at the gate level and actually called upon four times to perform a 16-bit operation. Classical stuck-at-zero and stuck-at-one faults were inserted in the gates of the four slices, and three ISP variables were used for the purpose of fault injection. These were 1) SLICE.NUMBER, which states which slice number is to be faulted (0,1,2,3); 2) GATE.NUMBER, which states which gate is to be faulted; and 3) STUCK.VALUE, which states at what value the specific gate should be stuck. After the ISP description, the fault injection process was begun. Throughout fault injection, only single faults were injected; multiple faults were avoided. Fault injection is done prior to program execution and lasts until the end of the simulation.

Three different system-level manifestations occurred as the result of the fault injection: 1) the program terminated with a wrong result, 2) the

program got a PDP-11/70 trap, and 3) the program executed for an indefinite period of time. Examples of these three manifestations are listed below according to their frequency of occurrence.

1) The program terminates with the wrong results.

This was the most frequently occurring manifestation. In contrast to functional-level fault injection in which the program very rarely terminated at the normal termination point, more than 50 percent of these program simulations terminated at the normal point. The fact that the program ended with wrong results needs a careful study and a careful comparison with a simulation run without faults. Usually, the execution time for the faulty program was the same as that approximately for the correct program.

Look at the example below excerpted from program 0412.

LOCATION	VALUE	VALUE	LABEL	INSTRUCTION	
1000	012706	001446	START :	MOV	#STACK,SP
1004	012746	001212		MOV	#STR,-(SP)

Assume that gate 19 (see Appendix B) in the second slice is stuck-at-one (i.e., change from a logical to an arithmetic operation). To fetch an instruction, the PC is first put into the leg B and a constant (2) is put into leg A. Leg B (PC) is loaded into VAR by using the ALU function F=B. Then, using the function A+B, the PC is incremented by two to point to the next word. The PC value is initially set to 1000. Since gate 19 is stuck-at-one in the first pass through the ALU, F=B changes to F=$(A+\overline{B})$ PLUS AB PLUS 1, and since it is the second-bit slice, the operation is

$$1000(8) = 001000000000(2)$$

Leg B for this second slice is now 0000. Leg A is two (for the PC increment), and the result of the second slice is

$$(A+\overline{B}) \text{ PLUS AB PLUS } 1 = (0000+1111) \text{ PLUS } 0000 \text{ plus } 1 = 0000.$$

Consequently, the result is 1000(8), which it should be.

In the second pass the PC goes through the ALU, and since A PLUS B is used for incrementing PC, no error occurs. (Remember all logical operations were changed to arithmetic operations, but not vice-versa). The instruction from location 1000 is fetched and SP (the stack pointer) gets the value of 1446. To fetch the second instruction, PC (now 1004) passes through the ALU to be loaded into the VAR. Again, the second set of four bits will be affected, or

$$1004 = 001000000100.$$

At this time the leg B for the second four bits is 0000(2), whereas the leg A is 0002(8) = 000000000010, and because $\overline{C}$=H, then $(A+\overline{B})$ PLUS AB = (0000+1111)PLUS(0000.0000) = (1111)PLUS(0000) = 1111 = 1111 which means VAR = 001011110100 = 1364. The PC is incremented by two and becomes 1006. At this point, location 1364 (the data part of the system) is zero. As a result, the program executes a HALT and stops. An interesting point here is the fact that if one only looked at the PC, one could not determine what exactly happened, because PC was incremented the way it should be and at the termination point was pointing to a valid instruction, which was not a halt instruction. This is very important and it indicates the dependency of fault manifestation, fault detection, and fault analysis on the architecture implementation.

2) The program gets a PDP-11/70 trap.

This case generally changes to one of the other two manifestations if the trap vectors are loaded and a trap handler is present. An unused instruction trap occurs more often than any other type of trap, mainly because the program attempts to execute the data part of a two- or three-word instruction. Since the analysis of this case requires a detailed study of the gate-level diagram and a large number of calculations, no example is presented here. Instead, a part of the simulation (program 0412) is listed below, along with the sequence of events.

LOCATION	VALUE	VALUE	VALUE	LABEL	INSTRUCTION	
1000	012706	001446		START :	MOV	#STACK,SP
1004	012746	001212			MOV	#STR,(SP)
1010	016746	000174			MOV	STRLEN,-(SP)
1014	012746	001242			MOV	#ARG,-(SP)
1020	016746	000214			MOV	ARGLEN,-(SP)
1024	012746	001206			MOV	#LOC,-(SP)
1030	004767	000002			JSR	PC,CHRSRC
1034	000000				HALT	
1036	022766	000000	000012	CHRSRC:	CMP	0,12(SP)
1044	001003				BNE	NONZER

Assume that gate 49 of the ALU is stuck-at-one. Program execution is as follows:

INITIAL VALUE OF PC =1000

PC	VAR	INSTRUCTION FETCHED	SP
1006	1000	012706	
1010			1442
1016	1010	016746	
1020	1012		1440
1026	1020	016746	
1030	1022		1436
1036	1030	004767	
1040	1032		1430
1040			
1042			
1040	1042	000012	

Note that the instructions at locations 1004, 1014, and 1024 are skipped and JSR is not executed correctly. It branches to the middle of CMP and tries to execute the data part of CMP as an instruction.

3) The program executes indefinitely.

In other words, the program execution literally lasts forever, or it takes much longer than the time a normal program execution would take. Refer to the execution of program 0412 below. The execution sequence is listed with regard to the previous example.

INITIAL PC = 1000

PC	VAR	INSTRUCTION FETCHED
1000	1002	001446
1000	1002	001446
1000	1002	001446
.	.	.
.	.	.
.	.	.

As in the case of functional-level fault injection, factors such as environment and system state affect the outcome.

d. <u>Results Compared to Functional-Level Fault Injection in Chapter 4</u>

The following are the comparisons between functional-level fault injections as discussed in Chapter 4 and gate-level fault injections presented in the previous chapters.

1) Both types of fault injection have the same classes of fault manifestation: 1) The program terminates with a wrong result, 2) the program gets a PDP-11/70 trap, 3) the program is in an infinite loop, or 4) the program ends with a right result. The last category is not shown for gate-level fault injections, since only fault injections for the ALU were performed.

2) The frequency of the manifestation classes are different for functional-level fault injections and gate-level fault injections. For example, the program with a wrong result at the functional level very rarely stops at the normal termination point, whereas for gate-level fault injections, the program stops at the normal point about 50 percent of the time.

3) Functional-level fault injections are more visible and consistent, and they affect almost every operation. Gate-level fault injections are not easily visible, they affect operations in a random fashion (pattern-sensitive).

4) Functional-level fault injections simulate pin faults very closely, whereas gate-level fault injections simulate die faults more accurately.

5) Fault injections at the functional level require only a functional level simulator. It is a higher level of abstraction, and the simulation time is relatively short. Gate-level simulations require a two-level simulator, and the simulation time increases considerably and is detailed.

6. RESULTS, SUMMARY, AND FURTHER WORK

a. Instructions As Fault Detectors

Simulation results indicate that an ALU instruction with a specific input detects, on the average, about 30 faulted gates stuck-at-one and 22 faulted gates stuck-at-zero. Further simulations show that for two specific ALU instructions and specific inputs for each of these functions, one can detect stuck-at-one faults for all but one of the gates. Similarly, stuck-at-zero faults will be detected for approximately 50 percent of the gates. The remaining 50 percent appear to be detectable by 2, 3 or, at most, 4 additional functions with specific inputs.

b. Fault Detection Circuit Considerations

Looking at the 70 percent of the total errors that are single-pin faults, one logically thinks parity is the error-detection scheme. The problem is that parity is not preserved under all the arithmetic/logic operations. To calculate what percentage of the total error is really caught by parity, the following solution is suggested. Consider a PLA with 14 inputs and 1 output in which six of the inputs are the ALU control lines, 4 are the A input to the ALU, and 4 are the B input to the ALU. The output of the PLA will be the parity-generated for the 4-bit output of the ALU. Using this scheme, there is a possible total of 362,143 errors. Of these, 311,656 are errors in the 4-bit output result of the ALU, and 50,487 are errors in $\overline{X}$, $\overline{Y}$ and $\overline{C}$. The total number of errors detected by the parity checker is 280,824, which is 78 percent (280824/362143) of total errors.

The next step is to generate parity for all 7 output bits from the ALU chip (i.e., 4-bit output result, $\overline{X}$, $\overline{Y}$ and $\overline{C}$). In this case, the total number of errors detected is 272,534, or 272534/362143 = 75%. There are several reasons why detection decreases. First, when parity is generated for 7 bits instead of 4 bits, the number of states in which an error can occur increases, but the parity of correct and faulted results are the same. Assume we are generating even parity. In the 4-bit case errors with the same parity can occur if the output has one or three bits on. In the 7-bit case the number of states with the same parity will be an output with

1, 3, 5 and 7 bits on. This increases the chance of missing an error. The second problem is that adding $\overline{X}$, $\overline{Y}$ and $\overline{C}$ bits will increase the probability of missing the errors since $\overline{C}$, $\overline{X}$ and $\overline{Y}$ will be in error in pairs most of the time; and can cause double-bit errors, which are undetectable.

Table 21 shows the result of generating parity for different combinations of output bits and the percentage of errors each can detect.

Table 21. Results of Generating Parity for Different Combinations of Output Bits.

Bits	Total No. of Errors	% Error Detected
$F_0F_1F_2F_3$	280824	78
$F_0F_1F_2F_3\overline{C}$	302583	83
$F_0F_1F_2F_3\overline{C}\ \overline{X}$	288435	80
$F_0F_1F_2F_3\overline{C}\ \overline{Y}$	292342	80
$F_0F_1F_2F_3\overline{C}\ \overline{X}\ \overline{Y}$	272534	75
$F_0F_1F_2F_3\overline{X}$	261016	72
$F_0F_1F_2F_3\overline{Y}$	288259	79
$F_1F_2F_3\overline{C}$	257285	71
$F_1F_2F_3\overline{C}\ \overline{Y}$	249518	68
$F_0F_1F_2F_3(\overline{X}$ or $\overline{Y}$ or $\overline{C})$	282222	77
$F_0F_1F_2F_3(\overline{X}$ or $\overline{C})$	277536	76
$F_0F_1F_2F_3(\overline{Y}$ or $\overline{C})$	283263	78
$F_0F_1F_2F_3\overline{C}\ (\overline{X}$ or $\overline{Y})$	294213	81

The table shows that the maximum percentage of errors (83) detected occurs by generating parity for $F_0F_1F_2F_3\overline{C}$.

If we choose a 14 by 2 PLA; i.e., generate parity for two sets of bits (i.e., $F_0F_1F_2F_3\overline{C}$ and $\overline{C}$, $\overline{X}$ or $\overline{Y}$), the total percentage increases to 319296/362145 = 88%. Even more than that, as Table 22 demonstrates.

Table 22. Total Percentage Increase When Generating Parity for Two Sets of Bits.

Bits	Total No. of Errors	% Error Detected
$F_0F_1F_2F_3\overline{C}$ & $\overline{C}\ \overline{X}\ \overline{Y}$	319296	88
$F_0F_1F_2F_3\overline{C}$ & $F_3\overline{C}\ \overline{X}\ \overline{Y}$	341390	94
$F_0F_1F_2F_3\overline{C}$ & $F_2F_3\overline{C}\ \overline{X}\ \overline{Y}$	337864	93

With 2 parity bits one is able to detect more than 94 percent of the errors. The remaining five or six percent can be detected by using the four, five, or, at most, six special ALU functions mentioned before. This can be done by periodically executing those special functions after each n instruction. Notice that the special function tests can be applied to all the 16 bits, whereas the parity checking is economical only if it is done for the 4 bits and then time-multiplexed for the other 4-bit slices, or if it is used in a round robin fashion for each 4-bit slice of ALU.

c. Level of BIT Verification

The availability of multiple implementations of a particular digital computer architecture makes it important to verify the system being used on a functional level without any real need for gate-level verification. RTI's approach to this problem was to use functional fault modeling and functional-level fault injection to predict accurately system-level manifestations of the faults. In addition, functional-level fault injection was used to represent pin faults if the fault continually exists throughout the program execution. In order to use functional-level fault injection for die (internal) faults, a very elaborate table is required which explains different manifestation dependencies in the input to the chip. (See Section 5b)

Fault injection itself is difficult because faults should only be injected for those inputs which cause the manifestation. Preparing the table and the way faults should be injected makes the functional-level fault injection unattractive. There is still a big problem associated with the functional-level fault injection related to the new technology. With

the advent of high-density, integrated circuits, manufacturers tend to put more BIT on a chip. If the purpose of functional-level fault injection is for BIT performance verification, one is completely ignoring the BIT on a chip. To understand this problem better, consider the following. Assume that C represents the set of chip faults, P represents the set of pin faults, and I represents the set of die or internal faults. Ignoring the effect of environmental transient faults and considering only the permanent faults $C = P \cup I$, assume that BIT is able to detect the B subset of the I set. Figure 24 shows that the only faults that are not detected are $P \cup (I \cap \overline{B})$. Now, since the functional-level fault injection is inserted at the functional and chip level, they are ignoring the effect of BIT because there is no actual fault for BIT to detect, whereas the manifestation of the fault exists at the functional level. From this discussion one can conclude that the BIT verification should be done at the level that the BIT is implemented. For example, if the BIT is implemented at the gate level, one should use the gate-level simulation or a two-level simulator (functional level with the capability of describing the modules at the gate level). If the BIT is implemented at the chip/module level, such as duplicate chip/module, one should use the chip/module-level simulation, etc.

d. Further Work

A great many complex issues pertaining to the use of machine-descriptive languages such as described in this report remain to be resolved. For example, the generality of the present case study needs to be determined. It is recommended that the proposed BIT performance verification approach be applied to the 32-bit architecture being evolved for MCF. As a part of such a study, it is proposed that special instructions be identified and developed for use in fault detection and isolation in future MCF embodiments.

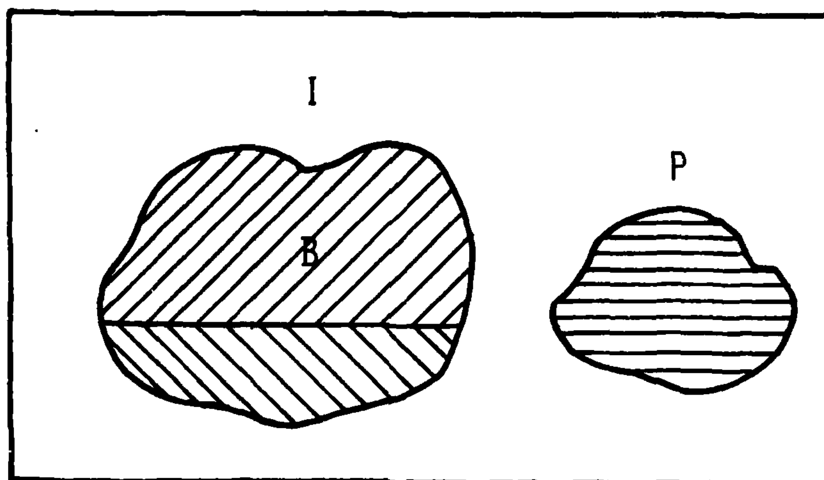


Figure 24. Functional Faults to Be Injected in the Presence of BIT.

7. REFERENCES

1. Bell, C. G. and A. Newell, Computer Structures: Readings and Examples, New York NY McGraw-Hill Book Company 1971.

2. Barbacci, M., "The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator," Technical Report. Department of Computer Science, Carnegie-Mellon University, 1976.

3. Barbacci, M. and D. Siewiorek, "Evaluation of the CFA Test Programs Via Formal Computer Descriptions," Computer, October 1977, pp. 36-43.

4. Clary, et al., "Built-In-Test Requirements for the Military Computer Family," Final Report, Contract No. DAAK80-79-C-0756, Prepared for CENTACS, Ft. Monmouth, NJ, June 1979.

5. Clary, J. B. and F. M. Smith, "Verification of Built-In-Test Performance in Modular Digital Systems Using Instruction Set Processor (ISP) Language Descriptions," Presented at AUTOTESTCON 1979, Minneapolis, MN, September 1979.

6. Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, "Architecture of the IBM System/360," IBM Journal of Research and Development, April 2, 1964.

7. PDP-11/70 Processor Handbook. Digital Equipment Corporation, Maynard, MA, 1975.

8. The TTL Data Book, Texas Instruments, Incorporated, 1976.

9. "Engineering Drawings for PDP-11/70," Digital Equipment Corporation, 1975.

10. Thatte, S. M. and J. A. Abraham, "Test Generation for General Microprocessor Architectures," Proc. 9th Annual International Symposium on Fault Tolerant Computing, June 1979, pp. 203-210.

11. Hosler, B., "Design and Implementation of a PMS Level Hardware Interconnection Language," Technical Report, Department of Computer Science, Carnegie-Mellon University, October 1979.

12. Northcutt, J. D., "High-Level Fault Insertion and Simulation with ISPS," Technical Report, Department of Computer Science, Carnegie-Mellon University, 1979.

13. Thatte, S. M. and J. A. Abraham, "A Methodology for Functional Level Testing of Microprocessors," Proc. 8th Annual International Conference on Fault-Tolerant Computing, June 1978, pp. 90-95.

14. Lai, L., "Functional Testing of Digital Systems," Thesis Proposal, Department of Computer Science, Carnegie-Mellon University, April 1978.

15. Snow, E. A. and D. P. Siewiorek, "Impact of Implementation Design Trade-offs on Performance: PDP-11, A Case Study," Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, February 1978.

16. Dietz, W. B. and L. Szewerenko, "Architectural Efficiency Measures: An Overview of Three Studies," Computer, April 1979, pp. 26-33.

APPENDICIES

APPENDIX A

(1) PDP-11/70 ISP Description

```
PDP11:=
BEGIN

**PDP11.70.PROCESSOR**
MACRO  TRUE  := | '1 | ,
MACRO  FALSE := | '0 | ,
MACRO  ON    := | '1 | ,
MACRO  OFF   := | '0 | ,
MACRO  SET   := | '1 | ,
MACRO  DONT.SET := | '0 | ,
MACRO  IN    := | '1 | ,
MACRO  OUT   := | '0 | ,

MACRO  WORD.SIZE  := | 15:0 | ,
MACRO  BYTE.SIZE  := |  7:0 | ,

MACRO READ.CNTRL:=| '01| ,
MACRO  WRITE.CNTRL:=| '10| ,
MACRO  DATA.LOW:=| 7:0| ,
MACRO DATA.HIGH:=| 15:8| ,


MACRO EQLU := | EQL(US]]| ,   ! UNSIGNED EQUAL OP
MACRO NEQU := | NEQ(US]]| ,
MACRO LSSU := | LSS(US]]| ,    ! UNSIGNED LSS OP
MACRO LEQU := | LEQ(US]]| ,
MACRO GEQU := | GEQ(US]]| ,
MACRO GTRU := | GTR(US]]| ,
**MP.STATE**
! MACRO DEFINITIONS TO ALLOW EASY CHANGE OF MEMORY CONFIGURATION.
! THE 11/70 ALLOWS ADDRESSING UP TO 2M * 2 BYTES   A SMALLER
! MEMORY IS DECLARED FOR SIMULATOR SPACE EFFICIENCY.
MACRO MAX.BYTE  := | #167777 | ,  ! (28K * 2 BYTES )
MB[MAX.BYTE:0]<7:0>,               ! THE ADDRESSING SPACE
MW[MAX.BYTE:0]<15:0>(INCREMENT:2]]:=MB[MAX.BYTE:0]<7:0>,
MBIO[#17777777:#17760000]<7:0>.           ! THE I/O PAGE (4K)
MWIO[#17777777:#17760000]<15:0>(INCREMENT:2]]:=MBIO[#17777777:#17760000]<7:0>,
**PC.STATE**
!R/REGISTER[15:0]<15:0>, ! REGISTER FILE INCLUDING TWO SETS OF GENERAL
                         ! REGISTERS: R0-R5 (ADDRESS 0000-0101, 1000-1101),
                         ! ONE PROGRAM COUNTER (ADDRESS 0111), AND THREE
                         ! STACK POINTERS (ADDRESS 0110,1110,1111)
    ! THE MAPPING IS AS FOLLOWS:
```


PRECEDING PAGE BLANK-NOT FILMED

```
!  CMODE (MODE)      RS (REG SET)    SRCREG OR DESREG        INDEX
!-----------------------------------------------------------------

!  XX                0                      000              0000
!  XX                0                      001              0001
!  XX                0                      010              0010
!  XX                0                      011              0011
!  XX                0                      100              0100
!  XX                0                      101              0101

!  XX                1                      000              1000
!  XX                1                      001              1001
!  XX                1                      010              1010
!  XX                1                      011              1011
!  XX                1                      100              1100
!  XX                1                      101              1101

!  00                X                      110              0110
!  01                X                      110              1110
!  11                X                      110              1111

!  XX                X                      111              0111

   ! X MEANS DOES NOT MATTER
MACRO PSW:= PS ,
PS<15:0>, ! := MBIO[#17777777:#17777776]<7:0>, PSW
       CM/CURRENT.MODE<1:0>     := PS<15:14>,   ! CURRENT ADDRESS SPACE
                                                ! (KERNEL/SUPERVISOR/USER)
       PM/PREVIOUS.MODE<1:0>    := PS<13:12>,   ! PREVIOUS ADDRESS SPACE
       P/PRIORITY<2:0>          := PS<7:5>,     ! CURRENT PROCESS PRIORITY
       RS/REGISTER.SET<>        := PS<11>,
       T/TRACE<>                := PS<4>,
       CC/CONDITION.CODES<3:0> := PS<3:0>,
              N/NEGATIVE<>             := CC<3>,
              Z/ZERO<>                 := CC<2>,
              V/OVERFLOW<>             := CC<1>,
              C/CARRY<>                := CC<0>,
SLR./STACK.LIMIT.REG<15:0>       := MBIO[#17777775:#17777774]<7:0>,
       SL/STACK.LIMIT.FIELD<7:0>       := SLR.<15:8>,
PIR/PROGRAM.INTERRUPT.REQUEST<15:0> := MBIO[#17777773:#17777772]<7:0>,
       PIA<2:0>         := PIR<3:1>,
INT.REQ.VEC[1:7]<15:0>,          !EXTERNAL INTERRUPT REQUEST VECTORS
ERRREG/CPU.ERROR.REGISTER<15:0>   := MBIO[#17777767:#17777766]<7:0>,
       ILLHLT/ILLEGAL.HALT<>             := ERRREG<7>,
       ODDADD/ODD.ADDRESS<>              := ERRREG<6>,
       NOMEM/NON.EXISTENT.MEMORY<>       := ERRREG<5>,
       TIMEOUT/UNIBUS.TIME.OUT<>         := ERRREG<4>,
       YELLOW/YELLOW.ZONE.STACK.LIMIT<>:= ERRREG<3>,
       RED/RED.ZONE.STACK.LIMIT<>        := ERRREG<2>,
```

```
MACRO KERNEL     := | (CM EQL '00) | ,
MACRO SUPER      := | (CM EQL '01) | ,
MACRO USER       := | (CM EQL '11) | ,
! TRAP VECTOR ADDRESSES:  THE ASSOCIATED ERROR CONDITIONS CAUSE EXECUTION
! TO SWITCH TO THE PC AND PS STORED IN THE TWO WORDS AT THE TRAP ADDRESS.
MACRO CPU.ERRORS        := | TRAP.VECTOR.ROM[0] | ,
MACRO ILL.INSTR         := | TRAP.VECTOR.ROM[1] | ,
MACRO RES.INSTR         := | TRAP.VECTOR.ROM[1] | ,
MACRO BPT.TRAP          := | TRAP.VECTOR.ROM[2] | ,
MACRO IOT.TRAP          := | TRAP.VECTOR.ROM[3] | ,
MACRO POWER.FAIL        := | TRAP.VECTOR.ROM[4] | ,
MACRO EMT.TRAP          := | TRAP.VECTOR.ROM[5] | ,
MACRO TRAP.TRAP         := | TRAP.VECTOR.ROM[6] | ,
MACRO PARITY            := | TRAP.VECTOR.ROM[7] | ,
MACRO PIR.TRAP          := | TRAP.VECTOR.ROM[8] | ,
MACRO FLT.TRAP/FLOATING.POINT   := | TRAP.VECTOR.ROM[9] | ,
MACRO MM.TRAP/MEMORY.MANAGEMENT := | TRAP.VECTOR.ROM[10] | ,
**MEMORY.MANAGEMENT**
! PAGE ADDRESS REGISTERS USED FOR ADDRESS MAPPING.
USEPAR/USER.PAGE.ADDRESS.REGISTER[15:0]<15:0>
                   :=MBIO[#17777677:#17777640]<7:0>,
KERPAR/KERNEL.PAGE.ADDRESS.REGISTER[15:0]<15:0>
                  :=MBIO[#17772377:#17772340]<7:0>,
SUPPAR/SUPER.PAGE.ADDRESS.REGISTER[15:0]<15:0>
                  :=MBIO[#17772277:#17772240]<7:0>,
! PAGE DESCRIPTION REGISTERS CONTAIN CONTROL INFORMATION ABOUT THE
! ADDRESS MAPPING.
USEPDR/USER.PAGE.DESCRIPTION.REGISTER[15:0]<15:0>
                  :=MBIO[#17777637:#17777600]<7:0>,
KERPDR/KERNEL.PAGE.DESCRIPTION.REGISTER[15:0]<15:0>
                  :=MBIO[#17772337:#17772300]<7:0>,
SUPPDR/SUPER.PAGE.DESCRIPTION.REGISTER[15:0]<15:0>
                  :=MBIO[#17772237:#17772200]<7:0>,
! VALID BITS IN THE PAGE ADDRESS REGISTER.
MACRO PAF/PAGE.ADDRESS.FIELD     := | 15:0 | ,
MACRO ACF/ACCESS.CONTROL.FIELD   := | 2:0 | ,
MACRO ED/EXPANSION.DIRECTION     := | 3 | ,
MACRO WBIT/WRITTEN.BIT           := | 6 | ,
MACRO PLF/PAGE.LENGTH.FIELD      := | 14:8 | ,
MACRO AIB/ACCESS.INFO.BITS       := | 7:6 | ,
MACRO ABIT/MET.ACF.CONDITIONS    := | 7 | ,
MMRO/STATUS.REGISTER.0<15:0>       := MBIO[#17777573:#17777572]<7:0>,
        ANR/ABORT.NONRESIDENT.FLAG<>     := MMRO<15>,
        APLE/ABORT.PAGE.LENGTH.FLAG<>    := MMRO<14>,
        ARO/ABORT.READ.ONLY.FLAG<>       := MMRO<13>,
        TMM/TRAP.MEMORY.MANAGEMENT<>     := MMRO<12>,
        EMMT/ENABLE.MM.TRAPS<>           := MMRO<9>,
        MDM/MAINTENANCE.DEST.MODE<>      := MMRO<8>,
        CIC/INSTRUCTION.COMPLETE<>       := MMRO<7>,
```

```
        AM/ABORT.MODE<1:0>                  := MMR0<6:5>,
        PAS/PAGE.ADDRESS.SPACE.I.D<>        := MMR0<4>,
        APN/ABORT.PAGE.NUMBER<2:0>          := MMR0<3:1>,
        EMM/ENABLE.MEMORY.MANAGEMENT<>      := MMR0<0>,

MMR1/STATUS.REGISTER.1<15:0>     := MBIO[#17777575:#17777574]<7:0>,
        ACHG2/AMOUNT.CHANGED.2<4:0>         := MMR1<15:11>,
        REGNO2/REGISTER.NUMBER.2<2:0>       := MMR1<10:8>,
        ACHG1/AMOUNT.CHANGED.1<4:0>         := MMR1<7:3>,
        REGNO1/REGISTER.NUMBER.1<2:0>       := MMR1<2:0>,
MMR2/STATUS.REGISTER.2<15:0>          := MBIO[#17777577:#17777576]<7:0>,
MMR3/STATUS.REGISTER.3<15:0>          := MBIO[#17772517:#17772516]<7:0>,
        EUM/ENABLE.UNIBUS.MAP<>             := MMR3<5>,
        E22M/ENABLE.22.BIT.MAPPING<>        := MMR3<4>,
        EKDS/ENABLE.KERNEL.D.SPACE<>        := MMR3<2>,
        ESDS/ENABLE.SUPER.D.SPACE<>         := MMR3<1>,
        EUDS/ENABLE.USER.D.SPACE<>          := MMR3<0>,
**IMPLEMENTATION.DECLARATIONS**
A.REG<WORD.SIZE>, ! REGISTER FIXED TO A.BUS, HOLDS OPERANDS
B.REG<WORD.SIZE>, ! REGISTER FIXED TO B.BUS, HOLDS ALU OPN

PLACE.HOLDER<16:0>,  ! HOLDS (REF]] DATA FROM DST.ADDR, LENGTH MISMATCH

A.LATCH<WORD.SIZE>,  ! REGISTER LATCH FOR THE ALU WHICH HOLDS A OPN
B.LATCH<WORD.SIZE>,   ! REGISTER LATCH  THE ALU WHICH HOLDS B OPN


ZEROS<63:0>,         ! ZERO FIELD


PWRF.SET<>,             ! POWER FAIL TRAP SET
FINT.SET<>,                         ! FLOATING POINT TRAP SET
BUS.REQUEST.VECTOR[7:4]<15:0>,      ! BUS REQUEST INTERRUPT VECTORS
                                    ! ADDRESS OF PC, PS INTERRUPT PAIR.


VAR/VIRTUAL.ADDRESS.REG<16:0>,

DR.IN/DATA.REGISTER.IN<WORD.SIZE>,
DR.OUT/DATA.REGISTER.OUT<WORD.SIZE>,


CONSTANT.ROM[0:5]<WORD.SIZE>,  !CONSTANTS 0,1,2,4,#31 AND ADDR. OF SLR

TRAP.VECTOR.ROM[0:15]<WORD.SIZE>, ! TRAP ADDRESSES
   TRAP.VECTOR<WORD.SIZE>,  ! OUTPUT REGISTER OF TRAP.VECTOR.ROM
**INSTRUCTION.FORMAT**
I/INSTRUCTION<15:0>,
BOP/BINARY.OPERATION<2:0>       := I<14:12>,
```

```
S/SOURCE.FIELD<5:0>                := I<11:6>,     ! SOURCE ADDRESS INFORMATION
        SRCMOD/SOURCE.MODE<2:0>            := S<5:3>,
        SRCREG/SOURCE.REG<2:0>             := S<2:0>,
D/DESTINATION.FIELD<5:0>           := I<5:0>,      ! DESTINATION ADDRESS INFO.
        DESMOD/DESTINATION.MODE<2:0>       := D<5:3>,
        DESREG/DESTINATION.REG<2:0>        := D<2:0>,
! INSTRUCTION DECODING FIELDS.
        UOP/UNARY.OPERATION<2:0>           := I<8:6>,
        OFFSET<7:0>                        := I<7:0>,
        ROP/REGISTER.OPERATION<1:0>        := I<7:6>,
        JETOP/JSR.EMULATOR.TRAP.OP<>       := I<15>,
        BYOP/BYTE.OPERATION<>              := I<15>,
        ETOP/EMULATOR.TRAP.OP<>            := I<8>,
        CONCOP/CONDITION.CODE.OP<10:0>     := I<15:5>,
        CPUOP/CPU.CONTROL.OP<2:0>          := I<2:0>,
        CONTOP/CPU.CONTROL.CLASS.OP<2:0>:= I<5:3>,
        BROP/BRANCH.OP.CODE<2:0>           := I<10:8>,
        INTOP/EXTENDED.INTEGER.OP<2:0>     := I<11:9>,
        TYPEOP/CLASS.OP.CODE.BITS<1:0>     := I<10:9>,
        RESOP/RESERVE.OP<>                 := I<11>,
        CCOP/CONDITION.CODE.SECOND.OP<>    := I<4>,
**DEFINITION.MACROS**

MACRO PSW.  := | #777777:#777776| , ! ADDRESS OF THE PSW
MACRO  PC.IN  := |  R(#7,'0) | ,
MACRO  PC.OUT:= |  R(#7,'1) | ,
MACRO  CONST0   := |  CONSTANT.ROM[0] | ,
MACRO  CONST1   := |  CONSTANT.ROM[1 ] | ,
MACRO  CONST2   := |  CONSTANT.ROM[2] | ,
MACRO  CONST4   := |  CONSTANT.ROM[3] | ,
MACRO CONST31   := |  CONSTANT.ROM[4]  | ,
MACRO SLR.ADDR  := |  CONSTANT.ROM[5]  | , ! THE ADDRESS OF SLR IN I/O PAGE
MACRO  SYSREG   := |  #777774:#777740  | ,  ! SYSTEM REGISTERS
MACRO  GPREG    := |  #777700:#777717  | ,
MACRO  USERPR   := |  #777600:#777677  | ,  ! USER PAGES
MACRO  MMR2.    := |  #777576:#777577  | ,
MACRO  MMR1.    := |  #777574:#777575  | ,
MACRO  MMR0.    := |  #777572:#777573  | ,
MACRO  MMR3.    := |  #772516:#772517  | ,
MACRO  KERPR    := |  #772300:#772377  | ,  ! KERNAL PAGES
MACRO  SUPR     := |  #772200:#772277  | ,  ! SUPERVISOR PAGES

! THESE MACROS ARE THE FUNCTIONS THAT THE ALU (74-181) WILL PERFORM,
! ACTUALLY ONLY A SUBSET OF ALL THE FUNCTIONS. THE NUMBERS ARE THE ACTUAL
! INPUTS NEEDED BY THE ALU TO DECODE TO THE PROPER FUNCTION.
MACRO TRNSF.A:=| #74| ,MACRO TRNSF.B:=| #52| ,MACRO CLR.:=| #14| ,
MACRO SUB.:=| #30| ,MACRO ADD.:=| #45| ,MACRO INC.:=| #00| ,MACRO DEC.:=| #75| ,
MACRO BIT.:=| #54| ,MACRO BIC.:=| #34| ,MACRO BIS.:=| #72| ,MACRO COM.:=| #02| ,
```

```
! THESE MACROS ARE JUST ARBITRARILY CHOSEN NUMBERS, CHOSEN SO THAT THEY
! DON'T USE ANY OF THE CURRENT ALU FUNCTION VALUES OR ANY OF THE OTHER
! ALU FUNCTION VALUES THAT WE MAY WANT TO USE LATER ON.
MACRO MOV.:=| #10| ,MACRO CMP.:=| #11| ,MACRO NEG.:=| #12| ,MACRO ADC.:=| #13| ,
MACRO SBC.:=| #20| ,MACRO TEST.:=| #21| ,MACRO SWAB.:=| #22| ,MACRO ROR.:=| #23| ,
MACRO ROL.:=| #24| ,MACRO ASR.:=| #25| ,MACRO ASL.:=| #26| ,MACRO MFP.:=| #27| ,
MACRO MTP.:=| #04| ,MACRO SXT.:=| #05| ,MACRO XOR.:=| #32| ,



**IMPLEMENTATION.MACROS**

MACRO  IR.D:=  | ZEROS@D@'0 | , ! SHIFT D FIELD IN INSTR.REG LEFT FOR MARK INSTR
MACRO  IR.CC:= | ZEROS@I<3:0> | , ! EXTRACTS CONDIT.CODES TO BE SET/CLEARED
MACRO  IR.CPUOP:= | '00000000@CPUOP@'00000 | , ! PRIORITY BITS FOR SPL
MACRO  IR.OFFSET:= | OFFSET@'0 | ,  ! BRANCH OFFSET, WITH SHIFT

MACRO SPL.PSW:= | PS<15:8>@'000@PS<4:0>| , !CHANGE THE PRIORITY LEVELS
MACRO RTI.RTT.PSW:= | PS<15:5>@'00000| ,

MACRO  POP.STACK:= | GET.OP(DR.IN,#2,#6,0) | , ! POPS THE STACK, POINTED TO BY SP
MACRO  PUSH( DATA ):= | PUT.OP(DATA,#4,#6,0) | , ! PUSH DATA ONTO STACK

MACRO SRC.OPN(REG,BYTE.OPN):= | GET.OP(REG,SRCMOD,SRCREG,BYTE.OPN)| ,
MACRO DST.OPN(REG,BYTE.OPN):= | GET.OP(REG,DESMOD,DESREG,BYTE.OPN)| ,
MACRO DST.ADDR(REG):= | GET.OP.ADDRESS(REG,DESMOD,DESREG,2)| ,
MACRO R.IN(INDX):= | R(INDX,'0) | ,
MACRO R.OUT(INDX):=| R(INDX,'1) | ,


! DEFINE FIELDS FOR SETTING APPROPRIATE CONDITION CODE BITS

MACRO  NZVC:= | '1111 | , ! SET/CLEAR ALL CC BITS
MACRO  NZV  := | '1110 | , ! SET/CLEAR N,Z,V, BITS DON'T MODIFY C
MACRO  NZ   := | '1100 | , ! SET/CLEAR N AND Z, DON'T MODIFY V OR C
MACRO  ZV   := | '0110 | , !SET/CLEAR Z AND V, DON'T MODIFY N OR C
MACRO  NVC  := | '1011 | ,
MACRO  NZC  := | '1101 | ,
MACRO  CV   := | '0011 | ,
MACRO  VS   := | '0010 | ,
MACRO  NS   := | '1000 | ,
MACRO  ZS   := | '0100 | ,

! MACRO USED TO CHECK THE VALIDITY OF THE INFORMATION PUT ON THE BUSSES
! THESE MACROS WILL BE EXPANDED IN THE FUTURE

MACRO  DATA.ERROR(SOURCE):= | FALSE | ,  ! JUST A TEMPORARY MACRO
```

```
MACRO  SET.ERROR.REGISTERS:= | NO.OP() | , ! JUST A TEMPORARY


MACRO  CHECK.BUS.AND.HANDLE.ERRORS:=
| IF DATA.ERROR( SOURCE ) =>
     BEGIN
       SET.ERROR.REGISTERS NEXT
       DECODE ICYCLE =>
          BEGIN
       '0/NOT.ICYCLE  := RESTART SERVICE, ! WERE IN SERVICE WHEN ERROR OCCURRED
       '1/ICYCLE.ACT := BEGIN  ! YOU WERE IN THE ICYCLE
                          ICYCLE_OFF NEXT   ! TURN OFF CYCLE SWITCH
                          LEAVE ICYCLE  ! GO TO SERVICE AND DEAL WITH ERROR
                        END
           END
        END | ,
**INITILIZATION.OF.ROMS**

INITIALIZE.ROMS:=
BEGIN
 TRAP.VECTOR.ROM[0]_#004; TRAP.VECTOR.ROM[1]_#010; TRAP.VECTOR.ROM[2]_#14;
 TRAP.VECTOR.ROM[3]_#20; TRAP.VECTOR.ROM[4]_#24; TRAP.VECTOR.ROM[5]_#30;
 TRAP.VECTOR.ROM[6]_#34; TRAP.VECTOR.ROM[7]_#114; TRAP.VECTOR.ROM[8]_#240;
 TRAP.VECTOR.ROM[9]_#244; TRAP.VECTOR.ROM[10]_#250;

 CONSTANT.ROM[0]_#0; CONSTANT.ROM[1]_#1; CONSTANT.ROM[2]_#2;
 CONSTANT.ROM[3]_#4; CONSTANT.ROM[4]=#16037; CONSTANT.ROM[5]=#177774

END,



**DATA.PATHS**
! THIS SECTION DEFINES THE DATA PATHS AND HOW THEY WORK.
! WORK IN THE SENSETHAT EVENTUALLY THESE PATHS WILL BE DEFINED
! WITH SOME FORM OF A CHECKER TO CHECK THEMSELVES.
! FOR NOW THE PATHS ARE MACROS, IN THIS WAY WE CAN REDUCE THE
! CODE GENERATED BY SIMPLE SUBSTITUTION.

MACRO A.BUS(SRC):= | SRC | ,
MACRO B.BUS(SRC):= | SRC | ,
MACRO C.BUS(SRC):= | SRC | ,

**MORE.ROUTINES**
! THIS IS A COLLECTION OF SUNDRY ROUTINES

! CC.SET SETS THE CONDITION CODE BITS IN THE PSW. TWO INPUTS A FOUR
! BIT FIELD EACH BIT REPRESENTS ONE CONDITION CODE BIT, IF THE CORRESPONDING
! BIT IS SET THEN THE CONDITION CODE BIT IS SET OR CLEARED DEPENDING ON
```

```
! THE STATE OR THE C.BUS.  THE SECOND INPUT SAYS WHETHER THIS IS A BYTE
! OR WORD ORIENTED INSTRUCTION. THIS IS USED BY THE CONDITION CODE
! GENERATOR FOR GENERATING THE NEW CC'S.AND THE THIRD INPU IS THE INSTRUCTION

CC.SET(BIT.SET<3:0>,BYTE<>,INSTR<5:0>):=
BEGIN
**LOCAL.MACROS**
MACRO N.BIT:=| BIT.SET<3>| ,
MACRO Z.BIT:=| BIT.SET<2>| ,
MACRO V.BIT:=| BIT.SET<1>| ,
MACRO C.BIT:=| BIT.SET<0>| ,

MAIN ENTRY.POINT:=
BEGIN

  IF N.BIT =>
      DECODE INSTR =>
      BEGIN
[MOV.,BIT.,BIC.,BIS.,ADD.,SUB.,COM.,TEST.,
 CMP.,INC.,DEC.,NEG.,ADC.,SBC.,ROL.,ASL.,ASR.]:=
        DECODE BYTE =>
        BEGIN
        0:= N=ALU<15>,
        1:= N=ALU<7>
        END,

 MFP.:= DECODE BYTE =>
        BEGIN
        0:= N=ALU<15>,
        1:= N=DR.IN<15>
        END,

 ROR.:=  N=B.LATCH<0>,  ! PREVIOUS C
 SWAB.:= N=ALU<7>,
 MTP.:= N=DR.IN<15>

      END NEXT

  IF Z.BIT =>
     DECODE INSTR =>
     BEGIN
[MOV.,CMP.,BIT.,BIC.,BIS.,ADD.,SUB.,COM.,INC.,DEC.,NEG.,ADC.,SBC.,
 TEST.,ROR.,ROL.,SXT.,CLR.,ASL.,ASR.]:=
        DECODE BYTE =>
        BEGIN
        0:= Z=ALU<15:0> EQLU 0,
        1:= Z=ALU<7:0> EQLU 0
        END,
 MFP.:= DECODE BYTE =>
```

```
            BEGIN
            0:= Z=ALU<15:0> EQLU 0,
            1:= Z=DR.IN<15:0> EQLU 0
            END,
    SWAB.:= Z=ALU<7:0> EQLU 0,
    MTP.:= Z= DR.IN<15:0> EQLU 0
          END NEXT

     IF C.BIT =>
        DECODE INSTR =>
        BEGIN
   [CMP.,ADD.,SUB.,ADC.,SBC.,ASL.,ASR.,ROR.,ROL.]:=
        C= ALU<16>,

    COM.:= C=1,
    NEG.:= C= NOT Z
        END NEXT



     IF V.BIT =>
        DECODE INSTR =>
        BEGIN
   [ROR.,ROL.,ASR.,ASL.]:=
           V=N XOR C,

   DEC.:= DECODE BYTE =>
           BEGIN
           0:= V=A.LATCH EQLU #100000,
           1:= V=A.LATCH<BYTE.SIZE> EQLU #200
           END,
   NEG.:= DECODE BYTE =>
           BEGIN
           0:= V=B.LATCH EQLU #100000,
           1:= V=B.LATCH<BYTE.SIZE> EQLU #200
           END,
    CMP.:=
           DECODE BYTE =>
           BEGIN
           0:= V=(ALU<15> EQV B.LATCH<15>) AND(A.LATCH<15> XOR B.LATCH<15>),
           1:= V=(ALU<7> EQV B.LATCH<7>) AND (A.LATCH<7> XOR B.LATCH<7>)
           END,

    ADD.:= V=(A.LATCH<15> EQV B.LATCH<15>) AND (A.LATCH<15> XOR ALU<15>),

    SUB.:= V=(ALU<15> EQV B.LATCH<15>) AND ( A.LATCH<15> XOR B.LATCH<15>),

    INC.:= DECODE BYTE =>
            BEGIN
```

```
        0:= V=ALU<15:0> EQLU #100000,
        1:= V=ALU<7:0> EQLU #200
        END,

 ADC.:= DECODE BYTE =>
        BEGIN
        0:= V=ALU<15> AND NOT A.LATCH<15>,
        1:= V=ALU<7> AND NOT A.LATCH<7>
        END,

 SBC.:= DECODE BYTE =>
        BEGIN
        0:= V=A.LATCH<15> AND NOT ALU<15>,
        1:= V=A.LATCH<7> AND NOT ALU<7>
        END
     END

END
END,

CC.CLR(BIT.CLR<3:0>):=
BEGIN
**LOCAL.MACROS**
MACRO N.BIT:= BIT.CLR<3> ,
MACRO Z.BIT:= BIT.CLR<2> ,
MACRO V.BIT:= BIT.CLR<1> ,
MACRO C.BIT:= BIT.CLR<0> ,

MAIN ENTRY.POINTS:=
BEGIN
  IF N.BIT => N=0;
  IF Z.BIT => Z=0;
  IF C.BIT => C=0;
  IF V.BIT => V=0
END
END,


GET.INDEX(CM<1:0>,REG<2:0>)<3:0>  :=
BEGIN
  DECODE REG =>
     BEGIN

[0:5]:= GET.INDEX_RS@REG,

   7:= GET.INDEX_'0111,

   6 := DECODE CM =>
          BEGIN
```

```
            '00:= GET.INDEX_'0110,
            '01:= GET.INDEX_'1110,
            '10:= STOP(),  ! ILLEGAL CM
            '11:= GET.INDEX_ '1111
              END
         END
    END,

    R(INDX<3:0>,R.W<>)<15:0>:=
    BEGIN
    **LOCAL.DEFN**
    REG[15:0]<15:0>,

    FAULT.SETTING<11:0>,
      INCL<>        :=FAULT.SETTING<11>, ! INCL
      MULT.SEL<>    :=FAULT.SETTING<10>,
      NO.SEL<>      :=FAULT.SETTING<9>,
      AND.OR<>      :=FAULT.SETTING<8>, ! AND=1, OR=0
      R.S/SELECTED.REG<7:0> :=FAULT.SETTING<7:0>,
    MACRO WRT:=| '0| ,
    MACRO RD:=| '1| ,

    MAIN REG.ENTRY:=
    BEGIN
      DECODE R.W =>
      BEGIN
    WRT:=  DECODE FAULT.SETTING<10:9> =>
            BEGIN
    '00:=   REG[INDX]=R,
    '01:=   IF (R.S SRO INDX)<0> EQLU '0 => REG[INDX]=R,
    '10:=   BEGIN
              IF R.S<0> => REG[0]=R;
              IF R.S<1> => REG[1]=R;
              IF R.S<2> => REG[2]=R;
              IF R.S<3> => REG[3]=R;
              IF R.S<4> => REG[4]=R;
              IF R.S<5> => REG[5]=R;
              IF R.S<6> => REG[6]=R;
              IF R.S<7> => REG[7]=R NEXT
              IF INCL => REG[INDX]=R
            END,
    '11:=   STOP()
           END,

    RD:=   DECODE FAULT.SETTING<10:9> =>
            BEGIN
    '00:=   R=REG[INDX],
    '01:=   IF (R.S SRO INDX)<0> EQLU '0 => R=REG[INDX],
    '10:=   DECODE AND.OR =>
```

```
          BEGIN
'0/OR:= BEGIN
          R<='0 NEXT
          IF R.S<0> => R=R OR REG[0];
          IF R.S<1> => R=R OR REG[1];
          IF R.S<2> => R=R OR REG[2];
          IF R.S<3> => R=R OR REG[3];
          IF R.S<4> => R=R OR REG[4];
          IF R.S<5> => R=R OR REG[5];
          IF R.S<6> => R=R OR REG[6];
          IF R.S<7> => R=R OR REG[7];
          IF INCL   => R=R OR REG[INDX]
        END,
'1/AND:=BEGIN
          R<='1 NEXT
          IF R.S<0> => R=R AND REG[0];
          IF R.S<1> => R=R AND REG[1];
          IF R.S<2> => R=R AND REG[2];
          IF R.S<3> => R=R AND REG[3];
          IF R.S<4> => R=R AND REG[4];
          IF R.S<5> => R=R AND REG[5];
          IF R.S<6> => R=R AND REG[6];
          IF R.S<7> => R=R AND REG[7];
          IF INCL   => R=R AND REG[INDX]
        END


          END,
'11:=   STOP()
        END
  END
END
END,
! GET.OP.ADDRESS COMPUTES MEMORY OPERAND ADDRESSES. IT CALLS
! READ TO COMPUTE INDEX AND DEFERRED ADDRESSES. ROUTINE SETMM1 IS CALLED
! IF A REGISTER IS INCR/DECR DURING ADDRESS CALCULATIONS TO SET MMR1.
! CHECK.STACK() IS ALSO CALLED IF THE REGISTER THAT IS INCR/DECR WAS THE SP
! GET.OP.ADDRESS IS MOSTLY CALLED BY GET.OP AND PUT.OP ALTHOUGH SOME
! INSTRUCTIONS USE IT DIRECTLY.
! A 17 BIT VIRTUAL ADDRESS IS COMPUTED WITH THE HIGH ORDER BIT BEING A
! SET TO ZERO FOR D SPACE AND A ONE FOR I SPACE.
! IF RESULT.REG IS NOT 17 BITS WIDE LIKE THE VAR THEN THE HIGH ORDER BIT
! IS LOST.  ALSO THE RESULT.REG MUST BE A DIRECT SINK OF THE C.BUS
! THE PARAMETERS ARE THE SAME AS GET.OP.

GET.OP.ADDRESS(RESULT.REG<16:0>(REF]],AMODE<2:0>,REG<2:0>,INCR<4:0>) :=
     BEGIN
      **LOCAL.MACRO**
      MACRO FORCE.WORD.INCR := | INCR=INCR +(US]] INCR<0> | ,
```

```
 MACRO WHAT.SPACE :=      | REG EQL #7 | , ! I SPACE IF REG=7
 MACRO D.SPACE    :=      | '0| ,        ! D SPACE
 MACRO I.SPACE    :=      | '1 | ,        ! I SPACE

MAIN ENTRY :=
      BEGIN
       GET.INDEX(CM,REG) NEXT

       DECODE AMODE=>
           BEGIN

              0:= (TRAP.VECTOR=ILL.INSTR NEXT
                    INSTR.TRAP() ),        ! REGISTERS HAVE NO ADDRESE
             1:=  BEGIN
                  A.LATCH_A.BUS(R.OUT(GET.INDEX)) NEXT
                  RESULT.REG_D.SPACE@C.BUS(  ALU(TRNSF.A,0)  )
                 END,

             2:=  BEGIN   ! AUTOINCREMENT
                  DECODE REG =>
                    BEGIN
                  6:=   FORCE.WORD.INCR,  ! SP EVEN INCREMENTS
                  7:=   INCR=2,          !  PC
                  OTHERWISE:= NO.OP()
                    END NEXT
                  A.LATCH_A.BUS(CONSTANT.ROM[INCR]); B.LATCH_B.BUS(R.OUT(GET.INDEX))
                               NEXT
                  RESULT.REG_C.BUS(  ALU(TRNSF.B,0)  );RESULT.REG<16>_WHAT.SPACE NEXT
                  R.IN(GET.INDEX)_C.BUS(  ALU(ADD.,0)  )NEXT
                  SETMM1(INCR,REG) ! SET MMR1 BY AMOUNT ADDED TO REG
                 END,

             3:= BEGIN   ! AUTOINCREMENT DEFFERRED
                  A.LATCH_A.BUS(CONST2);B.LATCH_B.BUS(R.OUT(GET.INDEX))
                               NEXT
                  VAR_C.BUS(  ALU(TRNSF.B,0)  ) ;VAR<16>_WHAT.SPACE NEXT
                  R.IN(GET.INDEX)_C.BUS(  ALU(ADD.,0)  ) NEXT
                  SETMM1(2,REG); DR.IN_READ(CM,0) NEXT
                  RESULT.REG_C.BUS(  DR.IN  )
                 END,

             4:= BEGIN    ! AUTODECREMENT
                  DECODE REG =>
                    BEGIN
                  6:=  FORCE.WORD.INCR,
                  7:=  UNDEFINED(),
                  OTHERWISE:=  NO.OP()
                    END NEXT
                  B.LATCH_B.BUS(CONSTANT.ROM[INCR]);A.LATCH_A.BUS(R.OUT(GET.INDEX))
```

```
                          NEXT
           RESULT.REG_R.IN(GET.INDEX)_C.BUS( ALU(SUB.,0)  ) NEXT
           SETMM1(-INCR,REG) NEXT
           IF REG EQL #6 => CHECK.STACK()
          END,

      5:= BEGIN    ! AUTODECREMENT DEFFERRED
                          NEXT
           R.IN(GET.INDEX)_VAR_D.SPACE@C.BUS(  ALU(SUB.,0)  );
                            NEXT
           DR.IN_READ(CM,0) NEXT
           SETMMI(-2,REG) NEXT
           IF REG EQL #6 => CHECK.STACK() NEXT
           RESULT.REG_D.SPACE@C.BUS( DR.IN )
          END,

      6:= BEGIN     ! INDEX

           A.LATCH_A.BUS(CONST2);B.LATCH_B.BUS(PC.OUT)
                          NEXT
           VAR_I.SPACE@C.BUS(  ALU(TRNSF.B,0)  )NEXT
           PC.IN_C.BUS(  ALU(ADD.,0)  );
           SETMMI(2,7) NEXT
           DR.IN_READ(CM,0)NEXT
           RESULT.REG_C.BUS(  DR.IN  )  NEXT
           A.LATCH_A.BUS(RESULT.REG);B.LATCH_B.BUS(R.OUT(GET.INDEX))

                          NEXT
           RESULT.REG_D.SPACE@C.BUS(  ALU(ADD.,0)  )
          END,

      7:= BEGIN     ! INDEX DEFFERRED
           A.LATCH_A.BUS(CONST2);B.LATCH_B.BUS(PC.OUT)
                          NEXT
           VAR_I.SPACE@C.BUS(  ALU(TRNSF.B,0)  )    NEXT
           DR.IN_READ(CM,0); PC.IN_C.BUS(  ALU(ADD.,0)  )  NEXT
           SETMMI(2,7); RESULT.REG_C.BUS(  DR.IN  )  NEXT
           A.LATCH_A.BUS(RESULT.REG);B.LATCH_B.BUS(R.OUT(GET.INDEX))
                          NEXT
           VAR_D.SPACE@C.BUS(  ALU(ADD.,0)  )  NEXT
           DR.IN_READ(CM,0)NEXT
           RESULT.REG_D.SPACE@C.BUS(  DR.IN  )
          END
      END
  END
END,
! ACCESS MEMORY OPERANDS. PARAMETERS ARE:
!       SINK REGISTER FOR THE C.BUS (REF]]
!       ADDRESS MODE (0:7)
```

```
!       REGISTER SPECIFIER (0:7)
!       OPERAND SIZE (BYTE OR WORD)
! THE BULK OF THE WORK IS DONE BY GET.OP.ADDRESS
GET.OP(RESULT.REG<15:0>(REF]],AMODE<2:0>,REG<2:0>,BYTE<>) :=
        BEGIN
        DECODE AMODE =>
                BEGIN
                0 :=    (A.LATCH_A.BUS(R.OUT(GET.INDEX(CM,REG))) NEXT
                           RESULT.REG_C.BUS(ALU( TRNSF.A,0 ))    ),
                OTHERWISE :=
                        BEGIN
                        GET.OP.ADDRESS(VAR,AMODE,REG,2-(US]] BYTE) NEXT
                          DR.IN_READ(CM,BYTE) NEXT
                        DECODE BYTE =>
                        BEGIN
                        '0:= RESULT.REG_C.BUS(DR.IN),
                        '1:= RESULT.REG_ZEROS<BYTE.SIZE>@C.BUS(DR.IN)<BYTE.SIZE>
                         END
                         END
             END
        END,
! PUT.OP STORES REGISTER OR MEMORY OPERANDS. IT INVOKES WRITE TO
! STORE MEMORY OPERANDS. PARAMETERS ARE:
!       ADDRESS MODE (0:7)
!       SOURCE REGISTER FOR THE C.BUS
!       REGISTER SPECIFIER (0:7)
!       OPERAND SIZE (BYTE OR WORD)
! THE BULK OF THE WORK IS DONE BY GET.OP.ADDRESS
PUT.OP(OUTPUT.REG<15:0>,AMODE<2:0>,REG<2:0>,BYTE<>) :=
        BEGIN
        DECODE AMODE =>
                BEGIN
                0 := BEGIN
                     R.OUT(GET.INDEX(CM,REG)) NEXT ! KLUDGE FOR BYTE OP
                     DECODE BYTE =>
                        BEGIN
                        R.IN(GET.INDEX) = C.BUS(  OUTPUT.REG  ),
                        R.IN(GET.INDEX)<BYTE.SIZE> = C.BUS( OUTPUT.REG )
                        END
                     END,
                OTHERWISE :=
                        BEGIN
                        DR.OUT_C.BUS(  OUTPUT.REG  )  NEXT
                        GET.OP.ADDRESS(VAR,AMODE,REG,2-(US]]BYTE) NEXT
                        WRITE(CM,BYTE)
                        END
                END
        END,
! REP.OP REPLACES REGISTER OR MEMORY OPERANDS. IT INVOKES WRITE TO
```

```
! STORE MEMORY OPERANDS. PARAMETERS ARE:
!       SOURCE REGISTER FOR C.BUS
!       ADDRESS MODE (0:7)
!       REGISTER SPECIFIER (0:7)
!       OPERAND SIZE (BYTE OR WORD)
! IT DIFFERS FROM PUT.OP IN THAT IT DOES NOT COMPUTE THE ADDRESS (AND
! ITS SIDE EFFECTS). IT USES THE CARRIER OF GET.OP.ADDRESS DIRECTLY
REP.OP(OUTPUT.REG<15:0>,AMODE<2:0>,REG<2:0>,BYTE<>) :=
        BEGIN
        DECODE AMODE =>
                BEGIN
                0 := BEGIN
                     R.OUT(GET.INDEX(CM,REG)) NEXT ! KLUDGE FOR BYTE OP
                     DECODE BYTE =>
                        BEGIN
                        R.IN(GET.INDEX) = C.BUS( OUTPUT.REG ),
                        R.IN(GET.INDEX)<BYTE.SIZE> = C.BUS( OUTPUT.REG )
                        END
                     END,
                OTHERWISE :=
                        BEGIN
                        DR.OUT_C.BUS(  OUTPUT.REG  ) NEXT
                        WRITE(CM,BYTE)
                        END
              END
        END,
**SERVICE.FACILITIES**
CHECK.STACK<> :=
        BEGIN
        CHECK.STACK _ 0 NEXT
        IF KERNEL =>
            DECODE R.OUT(6)<15:5> TST SL@#7 =>
                BEGIN
                    BEGIN       !RED VIOLATION
                        TRAP.RED = RED = 1;
                        A.LATCH_A.BUS(CONST4) NEXT
                        R.IN(6)_C.BUS( ALU(TRNSF.A,0) ) NEXT
                        RESTART_RUN
                    END,
                    CHECK.STACK = TRAP.YELLOW = YELLOW = 1,     !YELLOW VIOLATION
                    NO.OP()
                END
        END,
ODDERR/ODD.ADDRESS.ERROR :=
        BEGIN
        TRAP.CPU.ERR = ODDADD = 1 NEXT
        RESTART RUN
        END,
! THIS FUNCTION SETS MEMORY MANAGEMENT REGISTER 1 WHEN AN AUTO
```

```
! INCREMENT OR DECREMENT OF A REGISTER IS DONE.  THE INCR/DECR
! OF THE REGISTER IS PASSED IN CHANGE, AND THE REGISTER
! NUMBER IS PASSED IN BY REGNUM.
SETMM1(CHANGE<4:0>,REGNUM<2:0>) :=
        BEGIN
        IF MMR0<15:12> EQL 0 =>
                BEGIN
                DECODE MMR1<7:0> EQL 0 =>
                        BEGIN
                        0 := (MMR1<15:11> = CHANGE; MMR1<10:8> = REGNUM),
                        1 := (MMR1<7:3> = CHANGE; MMR1<2:0> = REGNUM)
                        END
                END
        END,
SETMM2(VECTOR<15:0>) :=
! THIS FUNCTION SETS MEMORY MANAGEMENT REGISTER 2 IF IT IS NOT
! LOCKED UP BY A PREVIOUS MEMORY MANAGEMENT ERROR CONDITION.
! IT IS CALLED FROM SERVICE.
        BEGIN
        IF MMR0<15:12> EQL 0 => MMR2 = VECTOR; CIC = 0
        END,
PRIORITY(REQVEC<7:1>)<2:0> :=          !RETURN BIT NUMBER OF HIGHEST PRIORITY
        BEGIN
        DECODE REQVEC =>
                BEGIN
                '1??????        := PRIORITY = 7,
                '01?????        := PRIORITY = 6,
                '001????        := PRIORITY = 5,
                '0001???        := PRIORITY = 4,
                '00001??        := PRIORITY = 3,
                '000001?        := PRIORITY = 2,
                '0000001        := PRIORITY = 1,
                OTHERWISE       := PRIORITY = 0
                END
        END,
!VIRT.PHY PERFORMS THE MAPPING OF VIRTUAL ADDRESSES INTO PHYSICAL ADDRESSES
!AND THE SEGREGATION OF PHYSICAL ADDRESSES INTO MEMORY AND I/O BUS ADDRESSES.
!INPUTS ARE:
!       MODE - KERNEL, SUPERVISOR,USER TO BE USED IN MAPPING
!       ADDRESS- A 17 BIT VIRTUAL ADDRESS. THE TOP BIT IS 1 FOR I SPACE
!                       AND 0 FOR D SPACE
!       WRITE- A FLAG SET TO 1 IF THE ACCESS IS A WRITE
!THE VALUE OF VIRT.PHY IS A 22 BIT PHYSICAL ADDRESS IN ALL CASES. ALL
!ADDRESSES GEQ #17000000 ARE I/O BUS ACCESSES TO THE BUS ADDRESS CONTAINED
!IN THE LEAST SIGNIFICANT 18 BITS.
VIRT.PHY(MODE<1:0>,ADDRESS<16:0>,WRITE<>)<21:0> :=
    BEGIN(US]]
        !THE DEFINITIONS BELOW ALLOW THE PAGE DESCRIPTOR REGISTERS
        !AND THE PAGE ADDRESS REGISTERS TO BE REFERENCED AS A SINGLE
```

```
    !REGISTER FILE DESPITE THE FACT THAT THE USER MODE REGISTERS
    !ARE ASSIGNED TO A SEPARATE AREA OF THE I/O ADDRESS SPACE
    ** LOCAL.DEFINITIONS **
    PDR[#2617:0]<15:0> := MBIO[#17777637:#17772200]<7:0>,
    PAR[#2617:0]<15:0> := MBIO[#17777677:#17772240]<7:0>,
    MACRO EDS/ENABLE.DATA.SPACE :=
         | (MMR3<2:0> SLD MODE)<2>| ,
    MACRO KERNEL.BASE := | #40| ,      !BASE ADDRESSES
    MACRO SUPERV.BASE := | 0| ,
    MACRO USER.BASE :=      | #2600| ,
    PR.INDX<10:0>,
ABORT(ERROR<2:0>,PAGE<3:0>) :=      ! MEMORY MANAGEMENT ABORT FUNCTION
    BEGIN
    IF MMRO<15:13> EQL 0 =>
            BEGIN
            AM = CM;
            APN = PAGE;
            PAS = PAGE<3>;
            MMRO<15:13> = MMRO<15:13> OR ERROR
            END NEXT
    TRAP.MM = 1;
    RESTART RUN                                 ! ABORT THE INSTRUCTION
    END,
MAIN ENTRY :=
    BEGIN
    DECODE EMM =>             !CHECK MAPPING ENABLED
        BEGIN
            VIRT.PHY <=(TC]] (ADDRESS<15:13> EQL #7)@ADDRESS<15:0>,
            BEGIN   !MAP THE ADDRESS
                DECODE EDS =>       !CHECK D SPACE ENABLED
                    BEGIN
                        PR.INDX = ADDRESS<15:13>,           !I SPACE ONLY
                        PR.INDX = (NOT ADDRESS<16>)@ADDRESS<15:13>  !I AND D SPACE
                    END NEXT
                DECODE MODE =>
                    BEGIN
                        PR.INDX = PR.INDX + KERNEL.BASE,
                        PR.INDX = PR.INDX + SUPERV.BASE,
                        ABORT(#6,PR.INDX),  !RESERVED
                        PR.INDX = PR.INDX + USER.BASE
                    END NEXT
    !CHECK PAGE LENGTH
                IF (PDR[PR.INDX]<PLF> TST ADDRESS<12:6>) EQL
                    PDR[PR.INDX]<ED>@'0 => ABORT(#2,PR.INDX) NEXT
    !CHECK ACCESS TYPE
                DECODE WRITE =>
                 BEGIN
                    DECODE PDR[PR.INDX]<ACF> =>     !READ OPERATION
                        BEGIN
```

```
                    [0,3,7] :=  ABORT(#4,PR.INDX),
                    [1,4] :=    (TRAP.MM = TMM = EMMT; PDR[PR.INDX]<ABIT> =1),
                OTHERWISE :=    NO.OP()
                            END,
                        DECODE PDR[PR.INDX]<ACF> =>     !WRITE OPERATION
                            BEGIN
                    [0,3,7] :=  ABORT(#4,PR.INDX),
                    [1,2] :=    ABORT(#1,PR.INDX),
                    [4,5] :=    (TRAP.MM = TMM = EMMT; PDR[PR.INDX]<ABIT:WBIT> = '11),
                OTHERWISE :=    PDR[PR.INDX]<WBIT> = 1
                            END
                        END NEXT
        !PERFORM THE MAPPING
                    VIRT.PHY = (PAR[PR.INDX] + ADDRESS<12:6>)@ADDRESS<5:0> NEXT
        !DEAL WITH 18 BIT MAPPING
                    IF NOT E22M => VIRT.PHY<21:18> <=(TC]] VIRT.PHY<17:13> EQL(US]] #37
                END
            END
        END
    END,
READ(MODE<1:0>,BYTE.ACCESS<>)<WORD.SIZE> :=
        BEGIN
        IF VAR<0> AND NOT BYTE.ACCESS => ODDERR() NEXT
        VIRT.PHY(MODE,VAR,0) NEXT
        DECODE VIRT.PHY<21:18> EQLU #17 =>
                BEGIN
                0 := DECODE BYTE.ACCESS =>
                     BEGIN
                     READ=MW[VIRT.PHY],
                     READ<=MB[VIRT.PHY]
                     END,
                1 := DECODE VIRT.PHY<17:0> =>
                        BEGIN
                     PSW.:= DECODE BYTE.ACCESS=>
                             BEGIN
                             READ=PS,
                             READ<=PS<BYTE.SIZE>
                             END,
                   GPREG  := DECODE BYTE.ACCESS =>
                             BEGIN
                             READ=R.OUT(VIRT.PHY<3:0>),
                             READ<=R.OUT(VIRT.PHY<3:0>)<BYTE.SIZE>
                             END,
                      OTHERWISE:= DECODE BYTE.ACCESS =>
                                         BEGIN
                                         READ = MWIO[VIRT.PHY],
                                         READ<= MBIO[VIRT.PHY]
                                        END
                        END
```

```
                  END
          END,
  WRITE(MODE<1:0>,BYTE.ACCESS<>) :=
          BEGIN
           IF VAR<0> AND NOT BYTE.ACCESS => ODDERR() NEXT
           VIRT.PHY(MODE,VAR,1) NEXT
           DECODE VIRT.PHY<21:18> EQLU #17 =>
                  BEGIN
                  0 := DECODE BYTE.ACCESS =>
                       BEGIN
                       MW[VIRT.PHY]=DR.OUT,
                       MB[VIRT.PHY]=DR.OUT
                       END,
                  1 := BEGIN
                          DECODE VIRT.PHY<17:0> =>              ! IO.PAGE
                                  BEGIN
                             PSW.:= DECODE BYTE.ACCESS=>
                                    BEGIN
                                    PS=DR.OUT,
                                    PS<BYTE.SIZE>=DR.OUT<BYTE.SIZE>
                                    END,
                            GPREG:= DECODE BYTE.ACCESS =>
                                    BEGIN
                                    R.IN(VIRT.PHY<3:0>)=DR.OUT,
                                    R.IN(VIRT.PHY<3:0>)<BYTE.SIZE>=DR.OUT
                                    END,
               [USERPR,KERPR,SUPR]:=BEGIN
                                    DECODE BYTE.ACCESS =>
                                    BEGIN
                                      MWIO[VIRT.PHY]=DR.OUT,
                                      MBIO[VIRT.PHY]=DR.OUT
                                    END NEXT
                                    VIRT.PHY<0>=VIRT.PHY<0> AND
                                                  NOT BYTE.ACCESS;
                                    VIRT.PHY<5>=0 NEXT
                                    MBIO[VIRT.PHY]<AIB>=0
                                    END,
                           #777772:=BEGIN   ! PIR
                                    DECODE BYTE.ACCESS =>
                                    BEGIN
                                      MWIO[VIRT.PHY]=DR.OUT,
                                      MBIO[VIRT.PHY]=DR.OUT
                                    END NEXT
                                    PIR<7:5>=PIR<3:1>=PRIORITY(PIR<15:9>)
                                    END,
                             OTHERWISE:= DECODE BYTE.ACCESS =>
                                         BEGIN
                                         MWIO[VIRT.PHY]=DR.OUT,
                                         MBIO[VIRT.PHY]=DR.OUT
```

```
                                        END
                                 END
                          END
                   END
            END,
      BRANCH(CONDITION<>) :=
            BEGIN
            IF CONDITION =>
                        BEGIN
                          A.LATCH<=A.BUS(IR.OFFSET); B.LATCH_B.BUS(PC.OUT) NEXT
                          ALU( ADD.,0 ) NEXT
                          PC.IN_C.BUS( ALU )
                        END
            END,
      ! INTERRUPT SERVICE ROUTINES
      BUS.RESET := (NO.OP()),
      ! SETMM2 IS A FLAG TO SET THE MM2 REG
      ! MMR2 IS SET IF NOT  LOCKED UP BY A PREVIOUS MEMORY
      ! MANAGEMENT ERROR CONDITION.

      INTVEC(CHK<>,SETMM2<>) := ! TRAP VECTOR SETUP
      BEGIN
      **INTERRUPT.RTNS**
      OLD.CM<1:0>:= ( OLD.CM=CM ),
      NEW.PM:=       (PM=OLD.CM ),

      MAIN INTRPT:=
            BEGIN
            !SAVE OLD PS AND PC IN TEMPORARIES
            B.LATCH_B.BUS( PC.OUT )NEXT
            A.REG_C.BUS( ALU( TRNSF.B,0)) NEXT  ! A.REG=OLD.PC
            B.LATCH_B.BUS( PS );  OLD.CM()  NEXT
            B.REG_C.BUS( ALU( TRNSF.B,0 ) ) NEXT  ! B.REG=OLD.PSW

      !  GET NEW.PC, NEW.PSW POINTED AT BY TRAP.VECTOR
            B.LATCH_B.BUS(TRAP.VECTOR) NEXT
            VAR_C.BUS( ALU(TRNSF.B,0) ) NEXT
            DR.IN_READ(0,0) NEXT
            PC.IN_C.BUS(DR.IN);

            B.LATCH_B.BUS(TRAP.VECTOR); A.LATCH_A.BUS(CONST2) NEXT
            VAR_C.BUS( ALU(ADD.,0) )NEXT  ! ADDR. OF NEW.PSW
            DR.IN_READ(0,0) NEXT !PSW
            PS_C.BUS( DR.IN ) NEXT  NEW.PM() NEXT


      ! PUSH OLD.PSW,OLD.PC ON THE STACK
            B.LATCH_B.BUS( B.REG ) NEXT
            DR.OUT_C.BUS( ALU(TRNSF.B,0) ) NEXT
```

```
          A.LATCH_A.BUS(R.OUT(GET.INDEX(CM,6))); B.LATCH_B.BUS(CONST2) NEXT
          R.IN(GET.INDEX)_VAR_C.BUS( ALU(SUB.,0) ) NEXT
          WRITE(CM,0) NEXT  ! PUSH(OLD.PSW)

         B.LATCH_B.BUS( CONST2 ); A.LATCH_A.BUS( R.OUT(GET.INDEX) ) NEXT
         R.IN(GET.INDEX)_VAR_C.BUS( ALU(SUB.,0) ) NEXT ! SP-2
           A.LATCH_A.BUS(A.REG) NEXT
           DR.OUT_C.BUS( ALU(TRNSF.A,0) ) NEXT
           WRITE(CM,0) NEXT  ! PUSH( OLD.PC )
            IF CHK => CHECK.STACK() NEXT !  SEE IF WE EXCEEDED THE
                                        ! STACK LIMIT WITH THESE 2 PUSHES

            IF SETMM2 =>
               BEGIN
                 IF MMR0<15:12> EQLU 0 => MMR2=TRAP.VECTOR ;
                                            CIC=0
               END  NEXT

         IF CHK AND CHECK.STACK => RESTART RUN  ! HANDLE OVERFLOW IMMEDIATELY

         END
  END,
  INSTR.TRAP() :=          ! RESERVED AND ILLEGAL
         BEGIN                                              ! OPCODE SERVICE
         INTVEC(1,DONT.SET) NEXT
         ICYCLE_OFF NEXT
         LEAVE ICYCLE
         END,
  TRAP.RED<>        := SERVICE<14>,
  TRAP.CPU.ERR<>    := SERVICE<13>,
  TRAP.PARITY<>     := SERVICE<12>,
  TRAP.MM<>         := SERVICE<11>,
  TRAP.YELLOW<>     := SERVICE<10>,
  TRAP.PF<>         := SERVICE<9>,
  TRAP.FP<>         := SERVICE<8>,
  TRAP.TRACE<>      := SERVICE<7>,
  INT.REQ[1:7]<>    := SERVICE<6:0>,
  SERVICE<14:0> :=          !INTERRUPT FLAGS AND SERVICE ROUTINE
         BEGIN
         DECODE (SERVICE OR PIR<15:9>) AND ("FFFF SLO P) =>  !MASK REQUESTS BY PROCESSOR PRIO
            BEGIN
     #0 :=       NO.OP(),
  '1??????????????:= BEGIN        !RED STACK VIOLATION
                     TRAP.RED = 0 NEXT
                     TRAP.VECTOR_CPU.ERRORS NEXT
                     INTVEC(0,DONT.SET)
                     END,
  '01?????????????:= BEGIN        !CPU ERROR
                     TRAP.CPU.ERR _ 0 NEXT
```

```
                        TRAP.VECTOR_CPU.ERRORS NEXT
                        INTVEC(1,DONT.SET)
                    END,
'001?????????????:= BEGIN          !PARITY ERROR
                        TRAP.PARITY_0 NEXT
                        TRAP.VECTOR_PARITY NEXT
                        INTVEC(1,SET)
                    END,
'0001????????????:= BEGIN     ! MEMORY MANAGEMENT
                        TRAP.MM_0 NEXT
                        TRAP.VECTOR_MM.TRAP  NEXT
                        INTVEC(1,DONT.SET)
                    END,
'00001???????????:= BEGIN          !YELLOW STACK VIOLATION
                        TRAP.YELLOW_0 NEXT
                        TRAP.VECTOR_CPU.ERRORS  NEXT
                        INTVEC(0,DONT.SET)
                    END,
'000001??????????:= BEGIN          !POWER FAIL
                        TRAP.PF_0 NEXT
                        TRAP.VECTOR_POWER.FAIL  NEXT
                        INTVEC(1,DONT.SET)
                    END,
'0000001?????????:= BEGIN          !FLOATING POINT
                        TRAP.FP_0 NEXT
                        TRAP.VECTOR_FLT.TRAP  NEXT
                        INTVEC(1,DONT.SET)
                    END,
'00000001????????:= BEGIN          !TRACE TRAP
                        TRAP.TRACE_0 NEXT
                        TRAP.VECTOR_BPT.TRAP  NEXT
                        INTVEC(1,DONT.SET)
                        END,
                    !
                    !BEGINNING OF MASKABLE INTERRUPTS
                    !
OTHERWISE:=         BEGIN   !MASKABLE INTERRUPTS
                        DECODE PRIORITY(SERVICE<6:0>) GTRU PIR<3:1> =>
                            BEGIN
                                BEGIN       !PROGRAM INTERRUPT REQUEST
                                    TRAP.VECTOR_PIR.TRAP NEXT
                                    INTVEC(1,SET)
                                END,
                                BEGIN       !EXTERNAL INTERRUPT
                                    INT.REQ[PRIORITY]_0 NEXT
                                    TRAP.VECTOR_INT.REQ.VEC[PRIORITY] NEXT
                                    INTVEC(1,SET)
                                END
                            END
```

```
                END
            END
        END,

! THIS ROUTINE DEFINES THE ALU AND ITS FUNCTIONS:
! TWO PARAMETERS: FUNCT= THE FUNCTION TO BE PERFORMED,
!                 BYOP= IS THE OPERATION BYTE/WORD.
! THE ALU LATCHS A.LATCH AND B.LATCH ARE DEFINED OUTSIDE OF THE ALU
! DESCRIPTION.  THE ALU IS IN TWO PARTS, THE FIRST PART IS DEFINED FROM
! THE 74181 SERIES OF ALUS AND DOES THE ARITHMETIC AND LOGIC OPERATIONS
! WE WILL ONLY USE THE SEGMENT OF THESE OPERATIONS THAT THE PDP11/70
! USES.  THE OTHER PART OF THE ALU IS THE SHIFT/ROTATE/SWAP WHICH WILL
! BE IMPLEMENTED IN SOME OTHER MANNER.
! THE ARITHMETIC OPERATIONS ARE DEFINED FOR BOTH BYTE AND WORD SIZE
! ARITHMETIC OPERATIONS.  THIS CAN BE DONE IN A 4-BIT SLICED ALU BY
! DEACTIVATING THE TWO HIGH ORDER SLICES AND RECONNECTING THE CARRY
! BIT.  I DIDN'T BOTHER DOING THIS TO THE OTHER OPERATIONS SINCE ONLY
! THE ARITHMETIC OPERATIONS ARE CONCERNED WITH THE SIGN BIT AND WHERE
! IT IS. ON THE OTHER OPERATIONS THE HIGH BYTE WILL BE ELEMINATED WHEN
! THE RESULT IS STORED, THOUGH CONCEPTUALLY ALL THE OPERATIONS SHOULD
! BE DONE ON ONLY THE BYTE.

ALU(FUNCT<5:0>,BYOP<>)<16:0>:= ! BIT 16 IS FOR THE CARRY BIT
BEGIN
**FAULT.DEFN**
FAULT<1:0>, ! SPECIAL FAULT TYPE
MASK<15:0>,  ! CONSTANT MASK

MAIN ALU.ENTRY:=
BEGIN (TC]]
   ALU=ZEROS<16:0> NEXT ! CLEAN UP DETAIL
DECODE FUNCT =>
BEGIN
  [#1,#74,#76,#77]:= ALU=A.LATCH,
  [#52,#53]:= ALU=B.LATCH,
  [#14,#16,#17]  := ALU=#0,


  #30     := DECODE BYOP =>
             BEGIN
             '0:=  ALU=A.LATCH-B.LATCH,
             '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>-B.LATCH<BYTE.SIZE>
             END,
  #45     := DECODE BYOP =>
             BEGIN
             '0:=  ALU=A.LATCH+B.LATCH,
             '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>+B.LATCH<BYTE.SIZE>
             END,
   #00 := DECODE BYOP =>
```

```
            BEGIN
            '0:=  ALU=A.LATCH + #1,
            '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>+ #1<BYTE.SIZE>
           END,
     #75 := DECODE BYOP =>
            BEGIN
            '0:=  ALU=A.LATCH-#01,
            '1:=  ALU<16>@ALU<BYTE.SIZE>=A.LATCH<BYTE.SIZE>-#1<BYTE.SIZE>
            END,

   [#54,#56,#57]  := ALU=A.LATCH AND B.LATCH,
   [#72,#73]    := ALU=A.LATCH OR B.LATCH,
   [#34,#36,#37]:=ALU= NOT A.LATCH AND  B.LATCH,
   #02  :=ALU=NOT A.LATCH,
   [#32,#33]  := ALU=A.LATCH XOR B.LATCH, ! EXCLUSIVE OR

   SWAB. := ALU=A.LATCH<7:0>@A.LATCH<15:8>,
   ROR.  := DECODE BYOP =>
            BEGIN
            '0:= ALU=(B.LATCH<0>@A.LATCH) SRR 1,
            '1:= ALU<16>@ALU<7:0>=(B.LATCH<0>@A.LATCH<7:0>) SRR 1
            END,
   ROL.  := DECODE BYOP  =>
            BEGIN
            '0:= ALU=A.LATCH@B.LATCH<0>,
            '1:= ALU<16>@ALU<7:0>=A.LATCH<7:0>@B.LATCH<0>
            END,
   ASR.  := DECODE BYOP =>
            BEGIN
            '0:=(ALU<15:0><=A.LATCH<15:1> ; ALU<16>=A.LATCH<0>),
            '1:=(ALU<7:0> <=A.LATCH<7:1>  ; ALU<16> =A.LATCH<0>)
            END,
  ASL. :=  DECODE BYOP =>
            BEGIN
            '0:= ALU=A.LATCH@'0,
            '1:=( ALU<7:0>= A.LATCH<6:0>@'0; ALU<16>=A.LATCH<7>)
            END,
! THESE FOLLOWING INSTRUCTIONS SHOULD NEVER BE USED BY THE CPU BUT
! IF THERE IS AN ERROR IN THE FUNCTION SOME OF THESE CAN BE EXECUTED

  #15  := ALU<='111, ! MINUS 1
  #31  := ALU= A.LATCH - B.LATCH -1,
  #35  := ALU= (A.LATCH AND NOT B.LATCH) - 1,
  #44  := ALU= (A.LATCH + B.LATCH) + 1,
  [#46:#47] := ALU= NOT ( A.LATCH XOR B.LATCH),
  #55  := ALU= ( A.LATCH AND B.LATCH) - 1,
  #71  := ALU= (A.LATCH OR NOT B.LATCH) + A.LATCH,
  #70  := ALU= (A.LATCH OR NOT B.LATCH) + 1,
  #51  := ALU= (A.LATCH OR NOT B.LATCH) + (A.LATCH AND B.LATCH),
```

```
 #50  := ALU= (A.LATCH OR NOT B.LATCH) + (A.LATCH AND B.LATCH) + 1

END NEXT
   DECODE FAULT =>
   BEGIN
0:= NO.OP(),
1:= ALU<15:0>=ALU<15:0> XOR MASK,
2:= ALU<15:0>=ALU<15:0> + MASK,
3:= ALU<15:0>=ALU<15:0>- MASK
   END

END
END,
**INSTRUCTION.EXECUTION**
! INITIALIZATION SEQUENCE


MAIN START :=
        BEGIN

        INITIALIZE.ROMS();

        SL=0;                   ! CLEAR STACK LIMIT
        PIR=0;                  ! CLEAR PROGRAM INTERRUPT REG
        MMRO=MMR3=0;            ! TURN MEMORY MANAGEMENT OFF
        ERRREG=0;                ! CLEAR ALL CPU ERRRORS
        ZEROS=0;
        RUN()
        END,

! MAIN RUN CYCLE OF THE ISP

RUN/INSTRUCTION.INTERPRETATION :=
        BEGIN
        SERVICE() NEXT
        ICYCLE() NEXT
        RESTART RUN
        END,


ICYCLE<> :=
        BEGIN
        ICYCLE_ON;   ! ICYCLE SWITCH INDICATOR
        TRAP.TRACE = T;
        B.LATCH_B.BUS(PC.OUT); A.LATCH_A.BUS(CONST2) NEXT
        VAR_C.BUS( ALU( TRNSF.B,0 ) ); VAR<16>_'1 NEXT
          IF MMRO<15:12> EQL 0 =>
             ( MMR2 = VAR<15:0>; MMR1=CIC=0 ) NEXT
```

```
        DR.IN_READ(CM,0);  PC.IN_C.BUS( ALU( ADD.,0 ) ) NEXT
        I_C.BUS( DR.IN ) NEXT
        EXEC() NEXT
        ICYCLE_OFF
        END,


EXEC/INSTRUCTION.EXECUTION :=
        BEGIN
        DECODE BOP =>
                BEGIN
                #0 := RESEROP(),                ! RESERVED OP CODE
                #1 := MOV(),            ! MOVE INSTRUCTION
                #2 := CMP(),            ! COMPARE INSTRUCTION
                #3 := BIT(),            ! BIT TEST INSTRUCTION
                #4 := BIC(),            ! BIT CLEAR INSTRUCTION
                #5 := BIS(),            ! BIT SET INSTRUCTION
                #6 := BEGIN                     ! ADD AND SUBTRACT
                        DECODE BYOP =>
                                BEGIN
                                '0 := ADD(),
                                '1 := SUB()
                                END
                        END,
                #7 := BEGIN                     ! EIS AND FP
                         TRAP.VECTOR=RES.INSTR NEXT
                         INSTR.TRAP()
                END
        END  NEXT
        IF MMR0<15:12> EQL 0 =>
                DECODE I =>
                        BEGIN           ! BPT, IOT, EMT, AND TRP DO NOT SET CIC
                        [3,4,#104000:#104777] := NO.OP(),
                        OTHERWISE := CIC = 1
                        END
        END,
RESEROP/RESERVE.OP.CODE :=
        BEGIN
        DECODE RESOP =>
                BEGIN
                0 := BRANOP(),
                1 := CLASSOP()
                END
        END,
BRANOP/BRANCH.OP.CODES :=
        BEGIN
        DECODE (JETOP @ BROP)<3:0> =>
                BEGIN
                #00 := REGOP(),                 ! REGISTER INSTRUCTION
```

```
                #01 := BRANCH('1),                        ! BRANCH (BR OP #00004)
                #02 := BNE := BRANCH(NOT Z),     ! BRANCH IF NOT EQUAL
                #03 := BEQ := BRANCH(Z),                  ! BRANCH IF EQUAL
                #04 := BGE := BRANCH(N EQV V),   ! BRANCH IF GTR OR EQUAL
                #05 := BLT := BRANCH(N XOR V),            ! BRANCH IF LESS THAN
                #06 :=BGT :=BRANCH(NOT(Z OR (N XOR V))),! BRANCH IF GREATER THAN
                #07 := BLE := BRANCH(Z OR (N XOR V)),! BRANCH IF LESS OR EQUAL
                #10 := BPL := BRANCH(NOT N),              ! BRANCH IF PLUS
                #11 := BMI := BRANCH(N),                  ! BRANCH IF MINUS
                #12 := BHI := BRANCH(NOT (C OR Z)),       ! BRANCH IF HIGH
                #13 := BLOS := BRANCH(C OR Z),            ! BRANCH IF LOWER OR SAME
                #14 := BVC := BRANCH(NOT V),              ! BRANCH IF OVERFLOW CLEAR
                #15 := BVS := BRANCH(V),                  ! BRANCH IF OVERFLOW SET
                #16 := BCC := BRANCH(NOT C),              ! BRANCH IF CARRY CLEAR
                #17 := BCS := BRANCH(C)          ! BRANCH IF CARRY SET
                END
        END,
    REGOP/REGISTER.OPERATIONS :=
        BEGIN
        DECODE ROP =>
                BEGIN
                0 :=    BEGIN
                        DECODE CONTOP =>
                        BEGIN
                            0:= BEGIN
                                DECODE CPUOP =>
                                        BEGIN
                                        #0 := HALT(),             ! HALT
                                        #1 := WAIT.(),            ! WAIT FOR INTERRUPT
                                        #2 := RTI.RTT(),  ! RETURN FROM INTERRUPT
                                        #3 := BPT(),              ! BREAKPOINT TRAP
                                        #4 := IOT(),              ! INPUT/OUTPUT TRAP
                                        #5 := RESET(),            ! RESET EXTERNAL BUS
                                        #6 := RTI.RTT(),          ! RETURN FROM TRAP
                                       #7 := (TRAP.VECTOR=RES.INSTR NEXT
                                            INSTR.TRAP()) ! UNUSED OP
                                        END
                                END,
                    OTHERWISE:= BEGIN
                                  TRAP.VECTOR=RES.INSTR NEXT
                                  INSTR.TRAP()
                                END
                            END
                        END,
                1 := JMP(),                               ! JUMP
                2 :=    BEGIN
                        DECODE CONTOP =>
                                BEGIN
                                #0 := RTS(),              ! RETURN FROM SUBROUTINE
```

```
                                #1:#2 :=(TRAP.VECTOR=RES.INSTR NEXT
                                      INSTR.TRAP() ),
                                #3 := SPL(),             ! SET PRIORITY LEVEL
                                #4:#7 := CCO()           ! CONDITION CODE OPS
                                END
                           END,
                      3 := SWAB()                        ! SWAP BYTES
                      END
                 END,
            CLASSOP/SECONDARY.DECODE.INTO.CLASSES :=
                 BEGIN
                 DECODE INTOP =>
                      BEGIN
                      #4:= SUBEMT(),          ! SUBROUTINE/EMULATOR TRAP
                      #5:= SINGLOP(),         ! SINGLE OPERAND CLASS
                      #6:= SHIFTOP(),         ! SHIFT OPERATORS
                      #7:= (TRAP.VECTOR=RES.INSTR NEXT INSTR.TRAP()),
                   #0:#3:= STOP() ! FUCK UP, THESE SHOULD HAVE BEEN DONE BEFORE
                      END
                 END,

            SUBEMT/SUBROUTINE.EMULATOR.TRAP.AND.TRAP.INSTRUCTIONS :=
                 BEGIN
                 DECODE JETOP =>
                      BEGIN
                      0 := JSR(),             ! JUMP TO SUBROUTINE
                      1 :=    BEGIN
                              DECODE I<8> =>  ! EMT OR TRAP
                                   BEGIN
                                   0 := EMT(),
                                   1 := TRAP()
                                   END
                              END
                      END
                 END,
            SINGLOP/SINGLE.OPERAND.INSTRUCTIONS :=
                 BEGIN
                 DECODE UOP =>
                      BEGIN
                      #0 := CLR(),    ! CLEAR/BYTE
                      #1 := COM(),    ! COMPLEMENT/BYTE
                      #2 := INC(),    ! INCREMENT/BYTE
                      #3 := DEC(),    ! DECREMENT/BYTE
                      #4 := NEG(),    ! NEGATE/BYTE
                      #5 := ADC(),    ! ADD CARRY/BYTE
                      #6 := SBC(),    ! SUBTRACT CARRY/BYTE
                      #7 := TEST()    ! TEST/BYTE
                      END
                 END,
```

```
SHIFTOP/SHIFT.INSTRUCTIONS :=
        BEGIN
        DECODE UOP =>
                BEGIN
                #0 := ROR(),     ! ROTATE RIGHT/BYTE
                #1 := ROL(),     ! ROTATE LEFT/BYTE
                #2 := ASR(),     ! ARITHMETIC SHIFT RIGHT/BYTE
                #3 := ASL(),     ! ARITHMETIC SHIFT LEFT/BYTE
                #4 := MARK(),    ! MARK
                #5 := MFP(),     ! MOVE FROM PREVIOUS INSTRUCTION
                #6 := MTP(),     ! MOVE TO PREVIOUS INSTRUCTION
                #7 := SXT()              ! SIGN EXTEND
                END
        END,
MOV :=  ! MOVE AND MOVE BYTE
                ! MOV OPCODE #01, MOVB OP CODE #11
        BEGIN
        SRC.OPN(A.REG,BYOP) NEXT A.LATCH_A.BUS(A.REG) NEXT
        ALU( TRNSF.A,BYOP ) NEXT
        CC.SET(NZ,BYOP,MOV.); CC.CLR(VS) NEXT
       DECODE DESMOD =>
         BEGIN
         '0:= ( IF BYOP EQLU '1 => ALU<15:8><=ALU<7> NEXT
                PUT.OP(ALU,DESMOD,DESREG,'0) ),
          OTHERWISE:= PUT.OP(ALU,DESMOD,DESREG,BYOP)
          END
        END,
CMP :=  ! COMPARE AND COMPARE BYTE
                ! CMP OP CODE #02,! CMPB OP CODE #12
        BEGIN
        SRC.OPN(A.REG,BYOP) NEXT DST.OPN(B.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS(B.REG) NEXT
         ALU(SUB.,BYOP)  NEXT    ! A-B
        CC.SET(NZVC,BYOP,CMP.)
        END,
BIT :=  ! BIT TEST AND BIT TEST BYTE
                ! BIT OP CODE #03, BITB OP CODE #13
        BEGIN
        SRC.OPN(A.REG,BYOP) NEXT DST.OPN(B.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS(B.REG) NEXT
         ALU(BIT.,BYOP)  NEXT
        CC.SET(NZ,BYOP,BIT.); CC.CLR(VS)
        END,
BIC :=  ! BIT CLEAR AND BIT CLEAR BYTE
                ! BIC OP CODE #04, BICB OP CODE #14
        BEGIN
        SRC.OPN(A.REG,BYOP) NEXT DST.OPN(B.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS(B.REG) NEXT
        ALU( BIC.,BYOP ) NEXT
```

```
        CC.SET(NZ,BYOP,BIC.);CC.CLR(VS) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,
BIS :=  ! BIT SET AND BIT SET BYTE
                ! BIS OP CODE #05, BISB OP CODE #15
        BEGIN
        SRC.OPN(A.REG,BYOP) NEXT DST.OPN(B.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS(B.REG) NEXT
        ALU( BIS.,BYOP ) NEXT
        CC.SET(NZ,BYOP,BIS.);CC.CLR(VS) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,

ADD :=  ! ADD, ADD OP CODE #06
        BEGIN
        SRC.OPN(A.REG,0) NEXT DST.OPN(B.REG,0) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS(B.REG) NEXT
        ALU( ADD.,0 ) NEXT
        REP.OP(ALU,DESMOD,DESREG,0); CC.SET(NZVC,0,ADD.)
        END,
SUB :=  ! SUBTRACT, SUB OP CODE # 16, B.LATCH-A.LATCH
        BEGIN
        SRC.OPN(B.REG,0) NEXT DST.OPN(A.REG,0) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS(B.REG) NEXT
        ALU( SUB.,0 ) NEXT
        REP.OP(ALU,DESMOD,DESREG,0); CC.SET(NZVC,0,SUB.)
        END,
! SUBROUTINE, EMULATOR TRAP, AND TRAP INSTRUCTION EXECUTION
JSR :=  ! JUMP TO SUBROUTINE, JMP OP CODE #004
        BEGIN
         DST.ADDR(PLACE.HOLDER) NEXT B.REG_PLACE.HOLDER  NEXT

            A.LATCH_A.BUS(R.OUT(GET.INDEX(CM,SRCREG))) NEXT  ! SP_ PUSH(REG)
            ALU(TRNSF.A,0) NEXT
            PUSH(ALU) NEXT

            B.LATCH_B.BUS(PC.OUT) NEXT  !  REG_PC
            ALU(TRNSF.B,0) NEXT
            R.IN(GET.INDEX(CM,SRCREG))=C.BUS(ALU) NEXT

            B.LATCH_B.BUS(B.REG) NEXT  !  PC_B.REG
            ALU(TRNSF.B,0) NEXT
            PC.IN_C.BUS( ALU )
        END,
EMT :=  ! EMULATOR TRAP OP CODES, EMT OP CODE #104000:#104377
        BEGIN
        TRAP.VECTOR=EMT.TRAP NEXT
        INTVEC(1,DONT.SET)
        END,
```

```
TRAP := ! TRAP OP CODES, TRAP OP CODE #104400:#104777
        BEGIN
        TRAP.VECTOR=TRAP.TRAP NEXT
        INTVEC(1,DONT.SET)
        END,
! SINGLE OPERAND INSTRUCTION EXECUTION
CLR :=  ! CLEAR AND CLEAR BYTE,
                ! CLR OP CODE #0050, CLRB OP CODE #1050
        BEGIN
        ALU(CLR.,BYOP) NEXT
        CC.SET(ZS,BYOP,CLR.);CC.CLR(NVC) NEXT
        PUT.OP(ALU,DESMOD,DESREG,BYOP)
        END,
COM :=  ! COMPLEMENT AND COMPLEMENT BYTE,
                ! COM OP CODE #0051, COMB OP CODE #1051
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT A.LATCH_A.BUS(A.REG) NEXT
        ALU( COM.,BYOP ) NEXT
        CC.SET(NZC,BYOP,COM.); CC.CLR(VS) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,
INC :=  ! INCREMENT AND INCREMENT BYTE,
                ! INC OP CODE #0052, INCB OP CODE #1052
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT   A.LATCH_A.BUS(A.REG) NEXT
        ALU( INC.,BYOP ) NEXT
        CC.SET(NZV,BYOP,INC.) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,
DEC :=  ! DECREMENT AND DECREMENT BYTE,
                ! DEC OP CODE #0053, DECB OP CODE #1053
        BEGIN
         DST.OPN(A.REG,BYOP) NEXT
         A.LATCH_A.BUS(A.REG) NEXT
         ALU( DEC.,BYOP ) NEXT
         REP.OP(ALU,DESMOD,DESREG,BYOP); CC.SET(NZV,BYOP,DEC.)
        END,
NEG :=  ! NEGATE AND NEGATE BYTE
                ! NEG OP CODE #0054, NEGB OP CODE #1054
        BEGIN
        DST.OPN(B.REG,BYOP) NEXT
        A.LATCH_A.BUS(CONSTO); B.LATCH_B.BUS(B.REG) NEXT
        ALU( SUB.,BYOP ) NEXT
        CC.SET(NZVC,BYOP,NEG.) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,
! SINGLE OPERAND INSTRUCTION EXECUTION (CONTINUED)
ADC :=  ! ADD CARRY AND ADD CARRY BYTE,
                ! ADC OP CODE #0055, ADCB OP CODE #1055
```

```
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG) NEXT  B.LATCH_B.BUS( C ) NEXT
        ALU( ADD.,BYOP ) NEXT
        CC.SET(NZVC,BYOP,ADC.) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,
SBC :=  ! SUBTRACT AND SUBTRACT CARRY BYTE,
                ! SBC OP CODE #0056, SBCB OP CODE #1056
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG) NEXT   B.LATCH_B.BUS( C ) NEXT
        ALU( SUB.,BYOP ) NEXT
        CC.SET(NZVC,BYOP,SBC.) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP)
        END,
TEST := ! TEST AND TEST BYTE,
                ! TST OP CODE #0057, TSTB OP CODE #1057
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG) NEXT
        ALU(TRNSF.A,BYOP) NEXT
        CC.SET(NZ,BYOP,TEST.); CC.CLR(CV)
        END,
! JUMP, SWAB EXECUTION AND REGISTER OPERATION DECODE
JMP :=          ! JUMP, JUMP OP CODE #0001
        BEGIN
          DST.ADDR(PLACE.HOLDER) NEXT PC.IN_PLACE.HOLDER
        END,

SWAB :=                 ! SWAP BYTES, SWAB OP CODE #0003
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG) NEXT
        ALU(  SWAB.,0  ) NEXT
        REP.OP(ALU,DESMOD,DESREG,0);
        CC.SET(NZ,1,SWAB.); ! MAKE A BYTE TO CHECK ONLY <7> AND <7:0>
        CC.CLR(CV)
        END,
ROR :=  ! ROTATE RIGHT AND ROTATE RIGHT BYTE,
                ! ROR OP CODE #0060, RORB OP CODE #1060
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS( C ) NEXT
        ALU( ROR.,BYOP ) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP);
        CC.SET(NZVC,BYOP,ROR.)
        END,
ROL :=  ! ROTATE LEFT AND ROTATE LEFT BYTE,
```

```
                ! ROL OP CODE #0061, ROLB OP CODE #1061
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG); B.LATCH_B.BUS( C ) NEXT
        ALU(  ROL.,BYOP  ) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP);
        CC.SET(NZVC,BYOP,ROL.)
        END,
ASR :=  ! ARITHMETIC SHIFT RIGHT AND ARITHMETIC SHIFT RIGHT BYTE,
                ! ASR OP CODE #0062, ASRB OP CODE #1062
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG) NEXT
        ALU(  ASR.,BYOP  ) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP);
        CC.SET(NZVC,BYOP,ASR.)
        END,

ASL :=  ! ARITHMETIC SHIFT LEFT AND ARITHMETIC SHIFT LEFT BYTE,
                ! ASL OP CODE #0063, ASLB OP CODE #1063
        BEGIN
        DST.OPN(A.REG,BYOP) NEXT
        A.LATCH_A.BUS(A.REG) ; B.LATCH_B.BUS( C ) NEXT
        ALU(  ASL.,BYOP  ) NEXT
        REP.OP(ALU,DESMOD,DESREG,BYOP);
        CC.SET(NZVC,BYOP,ASL.)
        END,
MARK := ! MARK AND UNUSED OP CODES, MARK OP CODE #0064
        BEGIN
        DECODE JETOP  =>
              BEGIN
              0:=BEGIN
                A.LATCH_A.BUS( IR.D  );  ! IR.D SHIFT LEFT 1
                B.LATCH_B.BUS(  PC.OUT )  NEXT
                ALU(ADD.,0) NEXT
                B.REG_VAR_C.BUS(  ALU  ) NEXT
                DR.IN_READ(CM,0);  A.LATCH_A.BUS(R.OUT(GET.INDEX(CM,5))) NEXT
                PC.IN_C.BUS(  ALU(TRNSF.A,0)  ) NEXT
                R.IN(GET.INDEX)_C.BUS(  DR.IN  ) NEXT
                B.LATCH_B.BUS(B.REG); A.LATCH_A.BUS(CONST2) NEXT
                R.IN(GET.INDEX(CM,6))_C.BUS(ALU( ADD.,0 ))
                END,

             1:= (TRAP.VECTOR=RES.INSTR NEXT INSTR.TRAP() )
              END
        END,
MFP :=  ! MOVE FROM PREVIOUS INSTRUCTION AND DATA SPACE,
                ! MFPI OP CODE #0065, MFPD OP CODE #1065
        BEGIN
```

```
            DECODE DESMOD NEQ 0 =>
                BEGIN
              0:=BEGIN
                A.LATCH_A.BUS(R.OUT(GET.INDEX(PM,DESREG))) NEXT  !  SP_ PUSH(DESREG)
                ALU(  TRNSF.A,BYOP  ) NEXT
                CC.SET(NZ,0,MFP.) NEXT
                PUSH(ALU)
                END,
              1:=BEGIN
                DST.ADDR( VAR ); VAR<16>_NOT BYOP NEXT ! VAR_DESTINATION
                DR.IN_READ(PM,0) NEXT
                CC.SET(NZ,1,MFP.) NEXT
                PUSH(DR.IN)  !  SP_PUSH(  C[DESTINATION] )
                END
            END  NEXT
         CC.CLR(VS)
         END,
   MTP :=  ! MOVE TO PREVIOUS INSTRUCTION AND DATA SPACE,
                   ! MTPI OP CODE #0066, MTPD OP CODE #1066
            BEGIN
            DECODE DESMOD NEQ 0 =>
                BEGIN
                  0:=BEGIN
                    POP.STACK NEXT
                    R.IN(GET.INDEX(PM,DESREG))_C.BUS(DR.IN);
                    CC.SET(NZ,0,MTP.); CC.CLR(VS)
                     END,
                  1:=BEGIN
                    POP.STACK NEXT
                    CC.SET(NZ,1,MTP.); CC.CLR(VS) NEXT
                    DR.OUT_C.BUS(DR.IN) NEXT
                    DST.ADDR( VAR ); VAR<16>_NOT BYOP NEXT ! VAR_DESTINATION
                    WRITE(PM,0)
                     END
                 END
             END,
   SXT :=  ! SIGN EXTEND AND UNUSED OP CODE,
                   ! SXT OP CODE #0067
            BEGIN
            DECODE JETOP =>
                    BEGIN
                    0 :=    BEGIN
                            B.LATCH_B.BUS( N ); A.LATCH_A.BUS(CONSTO) NEXT
                            ALU( SUB.,0 ) NEXT
                            PUT.OP(ALU,DESMOD,DESREG,0) ;
                            CC.SET(ZS,0,SXT.); CC.CLR(VS)
                            END,
                    1 := (TRAP.VECTOR=RES.INSTR NEXT INSTR.TRAP() )
                    END
```

```
        END,
CCO :=
! CONDITION CODE OPERATORS.  SELECTIVELY CLEARS OR SETS THE SPECIFIED
! CONDITION CODE.  THE ASSEMBLER RECOGNIZES THE MNEMONICS CLC, CLV, CLZ,
! CLN, CCC (FOR CLEAR ALL CONDITION CODES), SEC, SEV, SEZ, SEN, AND SCC.
! COMPOUND SETTING OR CLEARING IS ACCOMPLISHED BY ORING.
        BEGIN
        A.LATCH_A.BUS( IR.CC ); B.LATCH_B.BUS(PSW) NEXT
        DECODE CCOP =>
                BEGIN
                0 := ALU(BIC.,0), ! NOT I<3:0> AND PSW.CC
                1 := ALU( BIS.,0 ),  !  PSW.CC OR I<3:0>
                END  NEXT
        PSW_C.BUS( ALU )
        END,
! CPU CONTROL INSTRUCTION EXECUTION
HALT :=         ! HALT, HALT OP CODE #000000
        BEGIN
        DECODE CM =>
                BEGIN
                '00 := STOP(),          !KERNEL
                OTHERWISE := TRAP.CPU.ERR = ILLHLT = 1
                END
        END,
WAIT. :=        ! WAIT FOR INTERRUPT, WAIT OP CODE #000001
        BEGIN
        WAIT((SERVICE OR PIR<15:9>) SRO P)
        END,
! CPU CONTROL INSTRUCTION EXECUTION (CONTINUED)
BPT :=          ! BREAKPOINT TRAP,   BPT OP CODE #000003
        BEGIN
        TRAP.VECTOR=BPT.TRAP NEXT
        INTVEC(1,DONT.SET)
        END,
IOT :=          ! INPUT/OUTPUT TRAP, IOT OP CODE #000004
        BEGIN
        TRAP.VECTOR=IOT.TRAP NEXT
        INTVEC(1,DONT.SET)
        END,
RESET :=                ! RESET EXTERNAL BUS, RESET OP CODE #000005
        BEGIN
        IF CM EQL '00 =>
          BEGIN
            A.LATCH_A.BUS(SLR.ADDR) NEXT
            VAR_C.BUS(ALU( TRNSF.A,0 )) NEXT
            DR.OUT_C.BUS(ALU(CLR.,0)) NEXT
            WRITE(CM,0)
          END
        END,
```

(2) ALU Described in ISP at Gate Level

```
ALU(FUNCT<5:0>,BYOP<>)<16:0>:=
BEGIN
**FUNKADELIC**
  SLICE.NUMBER<2:0>,
  GATE.NUMBER<6:0>,
  STUCK.VALUE<>,
  FUNK<5:0>,
     NOT.CARRY<> :=FUNK<0>,
  FAULT<>,


LOGIC(funk<5:0>(ref),A<3:0>,B<3:0>,NUM<2:0>)<6:0>:=
BEGIN
**DECLS**
G[15:77]<>,
S2.M<>,
MC<>,

S3<> :=FUNK<5>,  A3<> :=A<3>,  B3<> :=B<3>,
S2<> :=FUNK<4>,  A2<> :=A<2>,  B2<> :=B<2>,
S1<> :=FUNK<3>,  A1<> :=A<1>,  B1<> :=B<1>,
S0<> :=FUNK<2>,  A0<> :=A<0>,  B0<> :=B<0>,
M<>  :=FUNK<1>,
NOT.CN<> :=FUNK<0>,
NOT.COUT<>:=FUNK<0>,
MACRO M.NOT:=| G[19]| ,

Y<>   :=LOGIC<6>,
X<>   :=LOGIC<5>,
EQ<>  :=LOGIC<4>,
F3<>  :=LOGIC<3>,
F2<>  :=LOGIC<2>,
F1<>  :=LOGIC<1>,
F0<>  :=LOGIC<0>,

MAIN LOGIC.ENTRY:=
BEGIN

! THIS THE SLICE TO FAULT

FAULT='0  NEXT ! NO FAULT THE SLICE
 IF NUM EQL SLICE.NUMBER => FAULT='1 NEXT

! LEVEL 1 GATES
G[15]=NOT B3;
G[16]=NOT B2;
G[17]=NOT B1;
G[18]=NOT B0;
G[19]=NOT M     NEXT
```

```
IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 2 GATES
G[20]=A3 AND S3 AND B3;
G[21]=A3 AND S2 AND G[15];
G[22]=S1 AND G[15];
G[23]=S0 AND B3;
G[24]=A3;
G[25]=B2 AND S3 AND A2;
G[26]=A2 AND S2 AND G[16];
G[27]=S1 AND G[16];
G[28]=S0 AND B2;
G[29]=A2;
G[30]=B1 AND S3 AND A1;
G[31]=A1 AND S2 AND G[17];
G[32]=S1 AND G[17];
G[33]=S0 AND B1;
G[34]=A1;
G[35]=B0 AND S3 AND A0;
G[36]=A0 AND S2 AND G[18];
G[37]=S1 AND G[18];
G[38]=S0 AND B0;
G[39]=A0                                NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 3 GATES
G[40]=NOT(G[20] OR G[21]);
G[41]=NOT(G[22] OR G[23] OR G[24]);
G[42]=NOT(G[25] OR G[26]);
G[43]=NOT(G[27] OR G[28] OR G[29]);
G[44]=NOT(G[30] OR G[31]);
G[45]=NOT(G[32] OR G[33] OR G[34]);
G[46]=NOT(G[35] OR G[36]);
G[47]=NOT(G[37] OR G[38] OR G[39])    NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 4 GATES
G[48]=G[40] XOR G[41];
G[49]=G[42] XOR G[43];
G[50]=G[44] XOR G[45];
G[51]=G[46] XOR G[47]                 NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 5 GATES
MC=M.NOT AND NOT.CN NEXT
```

```
G[58]=MC AND G[46] AND G[44] AND G[42];
G[59]=M.NOT AND G[47] AND G[42] AND G[44];
G[60]=M.NOT AND G[45] AND G[42];
G[61]=M.NOT AND G[43];
G[62]=MC AND G[46] AND G[44];
G[63]=G[47] AND M.NOT AND G[44];
G[64]=M.NOT AND G[45];
G[65]=MC AND G[46];
G[66]=M.NOT AND G[47];
G[67]=NOT (MC)                          NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 6 GATES
G[69]=NOT(G[58] OR G[59] OR G[60] OR G[61]);
G[70]=NOT(G[62] OR G[63] OR G[64]);
G[71]=NOT(G[65] OR G[66])            NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 7 GATES
G[73]=G[48] XOR G[69];
G[74]=G[49] XOR G[70];
G[75]=G[50] XOR G[71];
G[76]=G[51] XOR G[67]               NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! LEVEL 8
G[77]=G[73] AND G[74] AND G[75] AND G[76] NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

! CARRIES
G[52]=G[41];
G[53]=G[40] AND G[43];
G[54]=G[40] AND G[42] AND G[45];
G[55]=G[40] AND G[42] AND G[44] AND G[47];
G[56]=NOT(G[40] AND G[42] AND G[44] AND G[46] AND NOT.CN);
G[57]=NOT(G[40] AND G[42] AND G[44] AND G[46]) NEXT

IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

G[68]=NOT(G[52] OR G[53] OR G[54] OR G[55]) NEXT
 IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT

G[72]=(NOT G[56]) OR (NOT G[68]) NEXT
 IF FAULT => G[GATE.NUMBER]=STUCK.VALUE NEXT
```

```
Y=G[68];
X=G[57];
EQ=G[77];
F3=G[73];
F2=G[74];
F1=G[75];
F0=G[76];

NOT.COUT=G[72]

END
END,


! ALU


MAIN ALU.ENTRY:=
BEGIN
ALU<='0  NEXT
DECODE FUNCT =>
BEGIN
SWAB.   :=ALU=A.LATCH<7:0>@A.LATCH<15:8>,
ROR.    :=DECODE BYOP =>
          BEGIN
          '0:= ALU=(B.LATCH<0>@A.LATCH) SRR 1,
          '1:= ALU<16>@ALU<7:0>=(B.LATCH<0>@A.LATCH<7:0>) SRR 1
          END,
ROL.    :=DECODE BYOP =>
          BEGIN
          '0:= ALU=A.LATCH$DB$U.LATCH<0>,
          '1:= ALU<16>@ALU<7:0>=A.LATCH<7:0>@B.LATCH<0>,
          END,
ASR.    :=DECODE BYOP =>
          BEGIN
          '0:=(ALU<15:0><=A.LATCH<15:1>;ALU<16>=A.LATCH<0>),
          '1:=(ALU<7:0> <=A.LATCH<7:1> ;ALU<16>=A.LATCH<0>)
          END,
ASL.    :=DECODE BYOP =>
          BEGIN
          '0:= ALU=A.LATCH$D'$U0,
          '1:=(ALU<7:0>=A.LATCH<6:0>@'0; ALU<16>=A.LATCH<7>)
          END,

OTHERWISE:=
        BEGIN

FUNK=FUNCT NEXT  ! USING FUNK TO RETURN THE CARRY OUT TO NEXT STAGE
```

```
ALU<3:0>=LOGIC(FUNK,A.LATCH<3:0>,B.LATCH<3:0>,1)<3:0> NEXT
ALU<7:4>=LOGIC(FUNK,A.LATCH<7:4>,B.LATCH<7:4>,2)<3:0> NEXT

IF BYOP =>(ALU<16>=NOT NOT.CARRY NEXT LEAVE ALU) NEXT

ALU<11:8>=LOGIC(FUNK,A.LATCH<11:8>,B.LATCH<11:8>,3)<3:0> NEXT
ALU<15:12>=LOGIC(FUNK,A.LATCH<15:12>,B.LATCH<15:12>,4)<3:0> NEXT

ALU<16>=NOT NOT.CARRY    ! CARRY OUT
END
END
END
END,
```

(3) ISP Description of the Memory Controller for the Dual-CPU Configuration

```
MEMORY.CONTROL(BUS.A<41:0>,BUS.A.RDY<>,BUS.B<41:0>,BUS.B.RDY<>):=
BEGIN
**MEMORY.BUS.AND.MACROS**
    REQUIRE.ISP | MACRO.MCF[X361JC40] | ,
MACRO MATCH.ERROR:= | 41 | ,

ERROR.RTN:=(   NO.OP() ),

! CHECKER IS THE INTERFACE BETWEEN THE REPLICATED CPUS AND THE MEMORY.
! CHECKER FIRST CHECKS THE BUSSES FROM CPU.A AND CPU.B FOR EQUALITY
! IF THEY ARE NOT EQUAL, BITWISE, THEN A MATCH ERROR HAS OCCURRED.
! THE CPUS ARE NOT DOING THE SAME THING THIS IS REPRESENTED BY THE STATE
! OF THE BUS.  IF THE BUSSES ARE EQUAL THEN CHECKER MOVES A COPY TO
! MEMORY.BUS AND THEN REQUESTS MEMORY TO PROCESS THE BUS.  AFTER
! THE BUS HAS BEEN PROCESS AND ITS STATE CHANGED THEN CHECKER REPLICATES
! THE STATE OF MEMORY.BUS ONTO BOTH THE BUSSES FOR CPU.A AND CPU.B.
! THE BUS.RDY FLAGS ARE USED TO INDICATE WHEN A CPU HAS LOAD ITS
! BUS WITH SOME COMMAND AND INFORMATION FOR MEMORY.  CHECKER WAITS FOR
! BOTH BUSSES TO RAISE THEIR FLAG THEN IT DOES ITS JOB.

MAIN CHECKER:=
BEGIN
                WAIT(BUS.A.RDY AND BUS.B.RDY) NEXT
   IF  BUS.A EQLU BUS.B  =>
     BEGIN
       MEM.BUS=BUS.A NEXT
            MEMORY.ACCESS() NEXT   ! GO TO MEMORY AND PROCESS
       BUS.A=BUS.B=MEM.BUS NEXT ! REPLICATE THE MEMORY.BUS
       BUS.A.RDY=BUS.B.RDY=FALSE NEXT ! TURN OFF THE FLAGS
       RESTART CHECKER   ! WAIT FOR NEXT MEMORY REQUEST
     END NEXT
! IF THE BUSSES AREN'T THE SAME THEN AN ERROR HAS OCCURRED
   ERROR.RTN() NEXT
   BUS.A<MATCH.ERROR>=BUS.B<MATCH.ERROR>=TRUE NEXT
   BUS.A.RDY=BUS.B.RDY=FALSE NEXT
   RESTART  CHECKER
END,
 MEMORY.ACCESS:=
BEGIN  **MEMORY.AND.MEMORY.REGISTERS**
MACRO MAX.BYTE:= | #167777 | , ! 28K*2 BYTES
    MB[MAX.BYTE:0]<7:0>,     ! THE ADDRESSING SPACE
    MW[MAX.BYTE:0]<15:0>(INCREMENT:2]]:=MB[MAX.BYTE:0]<7:0>,
          MBR<15:0>,
           BMBR<7 :0>          :=MBR<7:0>,
          MAR<21:0>,
          ERROR.REG<>,
          R.W<1:0>, ! INDICATES WHETHER READ OR WRITE CYCLE
          BYTE.ADDR<>, ! READ/WRITE A BYTE OR A WORD
```

```
  MAIN MEMORY.ROUTINES:=
BEGIN
      ERROR.REG$DB$UYTE.ADDR$DM$UBR$DM$UAR$DR$U.W_MEM.BUS<41:0> NEXT
      DECODE R.W  =>  ! DEPENDING ON READ/WRITE BIT
        BEGIN
       ['10,'11]:=     DECODE BYTE.ADDR => ! WRITE BYTE OR WORD
                         BEGIN
                        '0/WORD:= MW[MAR]_MBR,
                        '1/BYTE:= MB[MAR]_BMBR
                         END,
      ['00,'01]:=     DECODE BYTE.ADDR =>
                         BEGIN
                        '0/WORD:=MBR_MW[MAR],
                        '1/BYTE:=MBR<=MB[MAR]
                       END
        END NEXT
          MEM.BUS_ERROR.REG$DB$UYTE.ADDR$DM$UBR$DM$UAR$DR$U.W  ! WRITE ON BUS
    END
   END
END
```

(4) Execution of Some of the PDP-11/70 Diagnostics

```
C-MU BATCH VERSION 6B(425)-5 STREAM 1 AT 07 34·30 ON THURSDAY, 3 APR 80
   DOING DSKB:VERIFY.LOG[X361JC40]_DSKB:VERIFY.CTL[X361JC40] UNDER [X361JC40]
   MAXIMUM RUNTIME 01:10:00
   ERROR LIMIT IS 99999

OTHER JOBS SAME PPN: 6
0734        03-APR-80       THUR
 .SET TIME 4201
 .SET WATCH DAY,RUN
 .SET HPQ 1
 .SET DSKPRI -1
 .RUN TEMP:SIM
 [7:34:35]
 ISP SIMULATOR V10.1
 GDB:A;ISPS COMPILER V5B(3)-7;DSK:PDP11S.ISP[X361JC40]; 2 APR 80;13:28:09;

 SEQUENTIAL SIMULATION? [YES]: YES
 TYPE HELP FOR HELP
 TYPE CC TO INTERRUPT SIMULATION LOOPS
 LATEST NEWS:  4 JAN 80

 DO V R%REG[7]$HALT
 READ TEST1
 ! NORMAL END: PC=1422
 >DECHO
 27 LINES READ
 START

 R%REG[ 7]= 001422

 SIMULATION COMPLETED
 RUN TIME(MILLISECONDS)=36017
 RTM OPS EXECUTED=28405


 READ TEST2
 ! NORMAL END: PC=1716
 >DECHO
 42 LINES READ
 START

 R%REG[ 7]= 001716

 SIMULATION COMPLETED
 RUN TIME(MILLISECONDS)=320867
 RTM OPS EXECUTED=258700


 READ TEST3
```

```
! NORMAL END: PC=1256
>DECHO
18 LINES READ
START

R%REG[ 7]= 001256

SIMULATION COMPLETED
RUN TIME(MILLISECONDS)=22400
RTM OPS EXECUTED=21073


READ TEST4
! NORMAL END: PC=1314
>DECHO
21 LINES READ
START

R%REG[ 7]= 001314

SIMULATION COMPLETED
RUN TIME(MILLISECONDS)=26300
RTM OPS EXECUTED=24647


READ TEST5
! NORMAL END: PC=1216
>DECHO
16 LINES READ
START

R%REG[ 7]= 001216

SIMULATION COMPLETED
RUN TIME(MILLISECONDS)=19217
RTM OPS EXECUTED=18188


READ TEST6
! NORMAL END: PC=1216
>DECHO
16 LINES READ
START

R%REG[ 7]= 001216

SIMULATION COMPLETED
RUN TIME(MILLISECONDS)=20434
RTM OPS EXECUTED=19242
```

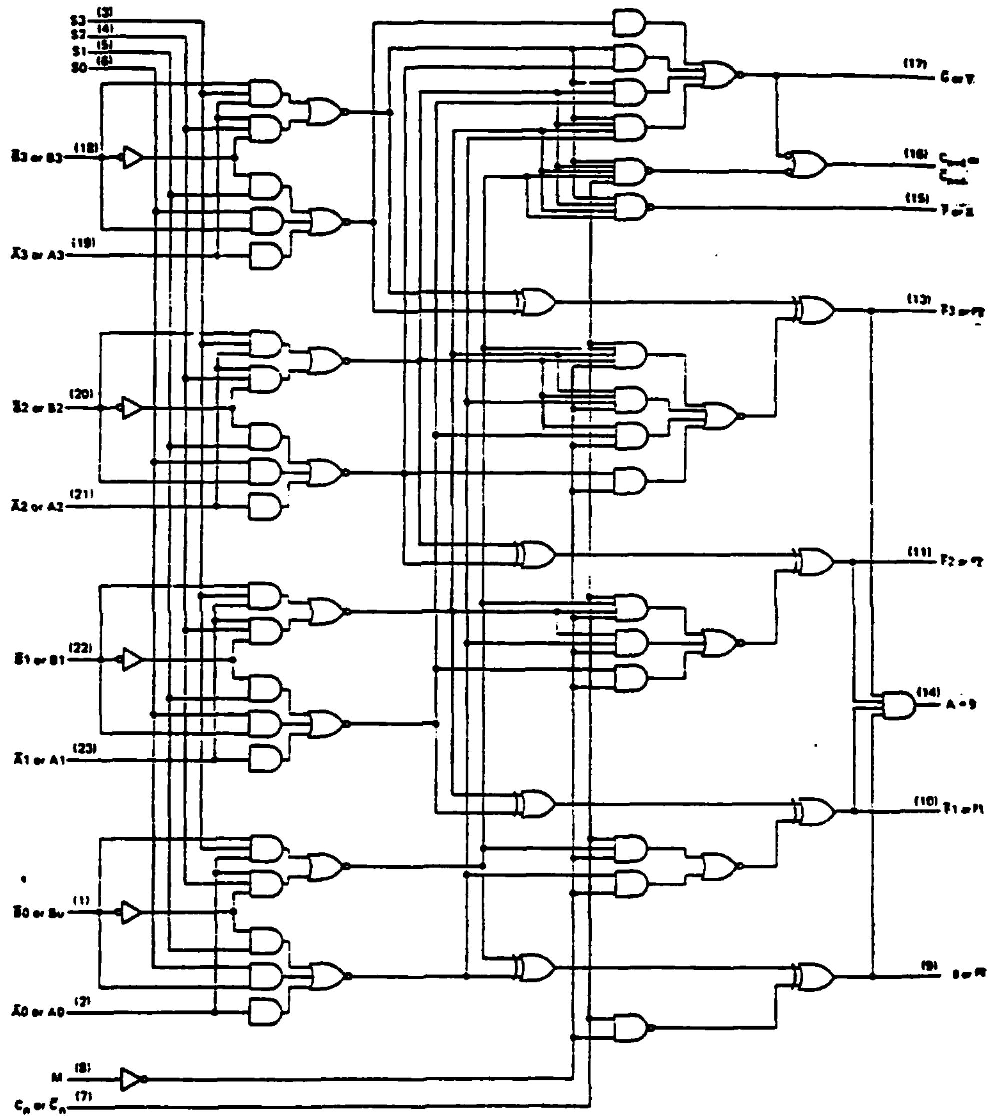
```
READ TEST7
 ! NORMAL END: PC=1256
>DECHO
18 LINES READ
START

R%REG[ 7]= 001256

SIMULATION COMPLETED
RUN TIME(MILLISECONDS)=20417
RTM OPS EXECUTED=19224
EXIT

EXIT
[:08:23   ]
.TIME
0.01
:08:23
KILO-CORE-SEC=36121
.SET TIME 30
.DEASSIGN
.KJOB/F
OTHER JOBS SAME PPN: 6
JOB 8, USER [X361JC40]  LOGGED OFF TTY121   0743  3-APR-80
ANOTHER JOB STILL LOGGED IN UNDER [X361JC40]
RUNTIME 8 MIN, 23.74 SEC; KILOCORE SEC: 36127
TOTAL OF 242 DISK BLOCKS READ, 6 WRITTEN
CONNECT TIME 0 HR, 9 MIN, 3 SEC; TOTAL CHARGE: $72.94
LOGGED OFF CMUD.
.
```

APPENDIX B - TI 74181 ALU

The modeling of the ALU functional fault is based upon a 74181 4-bit slice ALU. The reason for using this device is the easily accessible gate-level diagram for it, and also it is used in the PDP-11 family. This ALU has 6 control lines and two 4-bit inputs. The output of the ALU are 4-bit results, the carry bit and $\overline{X}$ and $\overline{Y}$ (carry generate and carry propagate). There are 48 different functions the TI 74181 ALU is able to perform. Figure B1 and Table B1 show the gate-level diagram and the functions performed by the TI 74181 ALU.

TYPES SN54181, SN54LS181, SN54S181, SN74181, SN74LS181, SN74S181
ARITHMETIC LOGIC UNITS/FUNCTION GENERATORS

functional block diagram



TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

Figure B1. TI 74181 ALU Logic Diagram.

Table B1. TI 74181 ALU Functions.

Selection				ACTIVE - HIGH DATA		
					M = L; Arithmetic Operations	
S3	S2	S1	S0	M = H Logic Functions	$\overline{C}_n$ = H (no carry)	$\overline{C}_n$ = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A plus 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) plus 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ plus 1
L	L	H	H	F = 0	F = minus 1 (2's compl)	F = Zero
L	H	L	L	$F = \overline{AB}$	$F = A$ plus $A\overline{B}$	$F = A$ plus $A\overline{B}$ plus 1
L	H	L	H	$F = \overline{B}$	F = (A + B) plus $A\overline{B}$	F = (A + B) plus $A\overline{B}$ plus 1
L	H	H	L	$F = A \oplus B$	F = A minus B minus 1	F = A minus B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ minus 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A plus AB	F = A plus AB plus 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A plus B	F = A plus B plus 1
H	L	H	L	F = B	$F = (A + \overline{B})$ plus AB	$F = (A + \overline{B})$ plus AB plus 1
H	L	H	H	F = AB	F = AB minus 1	F = AB
H	H	L	L	F = 1	F = A plus A*	F = A plus A plus 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) plus A	F = (A + B) plus A plus 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ plus A	$F = (A + \overline{B})$ plus A plus 1
H	H	H	H	F = A	F = A minus 1	F = A

APPENDIX C

(1) The Two CFA Candidate Programs

PROGRAM 0613

```
 1                        ;
 2                        ;TEST PROGRAM #13, PROGRAMMER #6,  MACHINE: PDP 11/70.
 3                        ;
 4                        ;         SIZE:   206 BYTES = 186 (CODE) + 20 (STACK)
 5                        ;
 6                        ;         COMPLETED:       28-JUL-79
 7                        ;
 8              000000            ROWPTR  = %0    ; INDEX INTO MASKS FOR CURRENT ROW BIT
 9              000001            COLPTR  = %1    ; INDEX INTO MASKS FOR CURRENT COL BIT
10              000002            ROWADR  = %2    ; ADDRESS OF CURRENT MATRIX ROW WORD
11              000003            COLADR  = %3    : ADDRESS OF CURRENT MATRIX COL WORD
12              000004            I       = %4    ; CURRENT SUB-MATRIX COUNTER
13              000005            J       = %5    ; CURRENT ROW/COL INDEX COUNTER
14              000006            SP      = %6
15              000007            PC      = %7
17              000022            ORDER   = 22    ; STACK OFFSETS FOR PARAMETERS
18              000022            BASE    = 22
19              000020            OFFSET  = 20

21              000040            MAXPTR  = 40    ; INDEX LIMIT INTO MASK VECTOR
22              000272'           TOPMASK = MASK + MAXPTR
23                                .GLOBL  BMT
24   000000' 010046               BMT:    MOV     ROWPTR,-(SP)     ; SAVE REGISTERS
25   000002' 010146                       MOV     COLPTR,-(SP)
26   000004' 010246                       MOV     ROWADR,-(SP)
27   000006' 010346                       MOV     COLADR,-(SP)
28   000010' 010446                       MOV     I,-(SP)
29   000012' 010546                       MOV     J,-(SP)

31   000014' 016646 000022                MOV     ORDER(SP),-(SP) ; FETCH PARAMETERS
32   000020' 006316                       ASL     @SP ; KEEP 2*ORDER BECAUSE MASK
                                                      ; INDICES ARE IN WORDS
33   000022' 016602 000022                MOV     BASE(SP),ROWADR : ADDRESS OF FIRST
                                                                ; MATRIX WORD
34   000026' 016600 000020                MOV     OFFSET(SP),ROWPTR : OFFSET OF FIRST
                                                                  ; BIT IN MATRIX

36   000034' 006300                       ASL     ROWPTR
```

```
37  000036' 062700 000232'          ADD    #MASK,ROWPTR

39  000042' 005004                  CLR    I        : INITIAL SUB-MATRIX COUNTER
40  000044' 000457                  BR     BEGIN    ; START TRANSPOSITION

42  000046' 060400          NEWROW: ADD    I,ROWPTR ; ADVANCE TO TOP-LEFT OF
                                                    ; SUB-MATRIX
43  000050' 020027 000272'  13$:    CMP    ROWPTR,#TOPMASK
44  000054' 002404                  BLT    1$
45  000056' 162700 000040           SUB    #MAXPTR,ROWPTR   ; ADVANCE TO NEXT
                                                            ; WORD OF BITS
46  000062' 005722                  TST    (ROWADR)+
47  000064' 000771                  BR     13$      ; CYCLE IN CASE INCREMENT
                                                    ; LARGER THAN ONE WORD
48  000066' 010405          1$:     MOV    I,J      ; SET UP INNER LOOP.
49  000070' 010203                  MOV    ROWADR,COLADR
50  000072' 010001                  MOV    ROWPTR,COLPTR

52  000074' 005720          DOBIT:  TST    (ROWPTR)+ ; ADVANCE TO NEXT BIT IN ROW
53  000076' 020027 000272'          CMP    ROWPTR,#TOPMASK
54  000102' 002403                  BLT    2$
55  000104' 012700 000232'          MOV    #MASK,ROWPTR     ; ADVANCE TO NEXT WORD
                                                            ; OF BITS
56  000110' 005722                  TST    (ROWADR)+
57  000112' 061601          2$:     ADD    @SP,COLPTR
58  000114' 020127 000272'  3$:     CMP    COLPTR,#TOPMASK
59  000120' 002404                  BLT    4$
60  000122' 162701 000040           SUB    #MAXPTR,COLPTR   ; ADVANCE TO NEXT WORD
                                                            ; OF BITS
61  000126' 005723                  TST    (COLADR)+
62  000130' 000771                  BR     3$       ; CYCLE IN CASE ORDER GREATER
                                                    : THAN  WORDLENGTH
63  000132' 005046          4$:     CLR    -(SP)    ; GET ROW BIT
64  000134' 031012                  BIT    @ROWPTR,@ROWADR
65  000136' 001401                  BEQ    5$
66  000140' 005216                  INC    @SP      ; WAS SET
67  000142' 005046          5$:     CLR    -(SP)    ; GET COLUMN BIT
68  000144' 031113                  BIT    @COLPTR,@COLADR
69  000146' 001401                  BEQ    6$
70  000150' 005216                  INC    @SP      ; WAS SET
71  000152' 005726          6$:     TST    (SP)+    ; STORE COLUMN BIT IN ROW
                                                    ; POSITION
72  000154' 001002                  BNE    7$
73  000156' 041012                  BIC    @ROWPTR,@ROWADR
74  000160' 000401                  BR     10$
75  000162' 051012          7$:     BIS    @ROWPTR,@ROWADR
76  000164' 005726          10$:    TST    (SP)+    ; STORE ROW BIT IN COLUMN
                                                    ; POSITION
77  000166' 001002                  BNE    11$
78  000170' 041113                  BIC    @COLPTR,@COLADR
```

```
79  000172' 000401                        BR      12$
80  000174' 051113               11$:    BIS     @COLPTR,@COLADR

82  000176' 005725               12$:    TST     (J)+     ; ADVANCE DOWN ROW/COL
83  000200' 020516                        CMP     J,@SP
84  000202' 002734                        BLT     DOBIT
85  000204' 005724               BEGIN:  TST     (I)+     ; ADVANCE TO NEXT SUB-MATRIX
86  000206' 020416                        CMP     I,@SP
87  000210' 002716                        BLT     NEWROW

89  000212' 005726                        TST     (SP)+    ; REMOVE LOCAL 'ORDER' VARIABLE

91  000214' 012605                        MOV     (SP)+,J  ; RESTORE REGISTERS
92  000216' 012604                        MOV     (SP)+,I
93  000220' 012603                        MOV     (SP)+,COLADR
94  000222' 012602                        MOV     (SP)+,ROWADR
95  000224' 012601                        MOV     (SP)+,COLPTR
96  000226' 012600                        MOV     (SP)+,ROWPTR
97  000230' 000207                        RTS     PC       ; SIGH!! ALL DONE.

99                                        ;       ------ BIT MASKS ------

101  000232' 000001          000002  000004  MASK:   .WORD   000001,000002,000004
102  000240' 000010          000020  000040          .WORD   000010,000020,000040
103  000246' 000100          000200  000400          .WORD   000100,000200,000400
104  000254' 001000          002000  004000          .WORD   001000,002000,004000
105  000262' 010000          020000  040000          .WORD   010000,020000,040000
106  000270' 100000                                  .WORD   100000
108                                          ;
109                                          ;       TEST PROGRAM SCAFFOLDING
110                                          ;
111              000000                      R0      = %0
112              000001                      R1      = %1
113  000272' 012706 000424'  START: MOV      #STACK,SP
114  000276' 012700 000424'         MOV      #TESTS,R0

116  000302' 012001                 MOV      (R0)+,R1

118  000304' 012046             NEXT:  MOV      (R0)+,-(SP)
119  000306' 012046                    MOV      (R0)+,-(SP)
120  000310' 012046                    MOV      (R0)+,-(SP)
121  000312' 004767 177462             JSR      PC,BMT
122  000316' 005301                    DEC      R1
123  000320' 001371                    BNE      NEXT

124  000322' 000000             HALT

126  000324' 000040                       .BLKW        40
127  000424'                              STACK:
129                                          ;
```

```
130                                            ;       TEST DATA
131                                            ;
132  000424' 000003                            TESTS:  .WORD   3 ; 3 TESTS ONLY

134                                            ;       TEST #1 (MEDIUM SIZED)

136  000426' 000004                                    .WORD   4 ; DIMENSION
137  000430' 000450'                                   .WORD   ARRAY1 ; ARRAY ADDRESS
138  000432' 000004                                    .WORD   4 ; OFFSET INTO FIRST
                                                               ; WORD

140                                            ;       TEST #2 (BIG TEST)

142  000434' 000011                                    .WORD   11 ; DIMENSION
143  000436' 000460'                                   .WORD   ARRAY2 ; ARRAY ADDRESS
144  000440' 000000                                    .WORD   0 ; OFFSET INTO FIRST
                                                               ; WORD

146                                            ;       TEST #3 (BOUNDARY TEST)

148  000442' 000001                                    .WORD   1 ; DIMENSION
149  000444' 000510'                                   .WORD   ARRAY3 ; ARRAY ADDRESS
150  000446' 000004                                    .WORD   4 ; OFFSET INTO FIRST
                                                               ; WORD

152                                            ;       ------ BIT MATRICES ------

154  000450' 016660          000011            ARRAY1: .WORD   016660,000011
155  000454' 020760          000013            WANT1:  .WORD   020760,000013

157                                                    .RADIX  2
158  000460' 151624                            ARRAY2: .WORD   1101001110010100
159  000462' 071035                                    .WORD   0111001000011101
160  000464' 101672                                    .WORD   1000001110111010
161  000466' 073516                                    .WORD   0111011101001110
162  000470' 174520                                    .WORD   1111100101010000
163  000472' 000000                                    .WORD   0

165  000474' 034526                            WANT2:  .WORD   0011100101010110
166  000476' 150664                                    .WORD   1101000110110100
167  000500' 053432                                    .WORD   0101011100011010
168  000502' 155266                                    .WORD   1101101010110110
169  000504' 044743                                    .WORD   0100100111100011
170  000506' 000000                                    .WORD   0

172                                                    .RADIX 8

174  000510' 000120                            ARRAY3: .WORD   000120

176   000272'                                          .END    START
```

PROGRAM 0412

```
1           000000                              R0 = %0
2           000001                              R1 = %1
3           000002                              R2 = %2
4           000003                              R3 = %3
5           000004                              R4 = %4
6           000005                              R5 = %5
7           000006                              SP = %6
8           000007                              PC = %7
9           000000'                             .CSECT MAINP
10  000000' 012706 000446'           START:     MOV     #STACK,SP
11  000004' 012746 000212'                      MOV     #STR,-(SP)
12  000010' 016746 000174                       MOV     STRLEN,-(SP)
13  000014' 012746 000242'                      MOV     #ARG,-(SP)
14  000020' 016746 000214                       MOV     ARGLEN,-(SP)
15  000024' 012746 000206'                      MOV     #LOC,-(SP)
16  000030' 004767 000002                       JSR     PC,CHRSRC
17  000034' 000000                              HALT
18  000036' 022766 000000  000012    CHRSRC:    CMP     #0,12(SP)
19  000044' 001003                              BNE     NONZER
20  000046' 005076 000002                       CLR     @2(SP)
21  000052' 000207                              RTS     PC
22  000054' 010046                   NONZER:    MOV     R0,-(SP)
23  000056' 010146                              MOV     R1,-(SP)
24  000060' 010246                              MOV     R2,-(SP)
25  000062' 010346                              MOV     R3,-(SP)
26  000064' 010446                              MOV     R4,-(SP)
27  000066' 010546                              MOV     R5,-(SP)
28  000070' 016600 000024                       MOV     24(SP),R0
29  000074' 166600 000020                       SUB     20(SP),R0
30  000100' 016601 000026                       MOV     26(SP),R1
31  000104' 016602 000022                       MOV     22(SP),R2
32  000110' 005200                              INC     R0
33  000112' 005300                   NOMTCH:    DEC     R0
34  000114' 002414                              BLT     NOTFND
35  000116' 121221                              CMPB    (R2),(R1)+
36  000120' 001374                              BNE     NOMTCH
37  000122' 010103                              MOV     R1,R3
38  000124' 010204                              MOV     R2,R4
39  000126' 005204                              INC     R4
40  000130' 016605 000020                       MOV     20(SP),R5
```

```
41  000134' 005305                   MATCH:  DEC     R5
42  000136' 003407                           BLE     FOUND
43  000140' 122423                           CMPB    (R4)+,(R3)+
44  000142' 001774                           BEQ     MATCH
45  000144' 000762                           BR      NOMTCH
46  000146' 012776 177777  000016   NOTFND: MOV     #-1,@16(SP)
47  000154' 000405                           BR      EXIT
48  000156' 005301                   FOUND:  DEC     R1
49  000160' 166601 000026                    SUB     26(SP),R1
50  000164' 010176 000016                    MOV     R1,@16(SP)
51  000170' 012605                   EXIT:   MOV     (SP)+,R5
52  000172' 012604                           MOV     (SP)+,R4
53  000174' 012603                           MOV     (SP)+,R3
54  000176' 012602                           MOV     (SP)+,R2
55  000200' 012601                           MOV     (SP)+,R1
56  000202' 012600                           MOV     (SP)+,R0
57  000204' 000207                           RTS     PC
58  000206' 177777                   LOC:    .WORD   -1
59  000210' 000026                   STRLEN: .WORD   26
60  000212'    115    157      156   STR:    .ASCII  /MONDAY, JUNE 7TH, 1976/
    000215'    144        141        171
    000220'    054        040        112
    000223'    165        156        145
    000226'    040        067        164
    000231'    150        054        040
    000234'    061        071        067
    000237'    066
61  000240' 000003                   ARGLEN: .WORD   3
62  000242'    144    141      171   ARG:    .ASCII  /DAY/
63  000245' 000100                           .BLKW   100
64  000446'                                  .EVEN
65  000446' 000000                   STACK:  .WORD   0
66   000000'                                 .END START
```

(2) Functional Fault Injection in the Instruction Decode and Control Functional Module

PROGRAM	FAULT	RESULT	COMMENTS
0613	DEC--NOP	INFINITE LOOP	THE LOOP CONTROL PARAMETER DOES NOT CHANGE
0613	ADD--NOP	HALT	2W--NOP,BR STOPS AT THE END OF THE PROGRAM
0613	SUB--NOP	TRAP	2W--NOP,NONE UNUSED INSTRUCTION TRAP
0613	CLR--NOP	TRAP	1W--NOP ODD ADDRESS TRAP
0613	BIT--NOP	INFINITE LOOP	1W--NOP
0613	JSR--NOP	TRAP	2W--NOP,NONE UNUSED INSTRUCTION TRAP
0613	CMP--NOP	HALT	2W--NOP,1W WRONG RESULT, ENDS FASTER
0613	ASL--NOP	TRAP	1W--NOP ODD ADDRESS TRAP
0613	BR--NOP	INFINITE LOOP	1W--NOP HAS NOT FINISHED FOR A WHILE
0613	BLT--NOP	INFINITE LOOP	1W--NOP HAS NOT FINISHED FOR A WHILE
0613	TST--NOP	TRAP	1W--NOP ODD ADDRESS
0412	JSR--NOP	TRAP	2W--NOP,1W FLOATING POINT INSTRUCTION TRAP
0412	CMP--NOP	HALT	3W--NOP,1W,1W ENDS IN THE MIDDLE OF PROGRAM
0412	BNE--NOP	HALT	1W--NOP ENDS IN THE MIDDLE OF PROGRAM
0412	RTS--NOP	TRAP	1W--NOP FLOATING POINT INSTRUCTION TRAP

PROGRAM	FAULT	RESULT	COMMENTS
0412	SUB--NOP	TRAP	2W--NOP,1W UNUSED INSTRUCTION TRAP
0412	INC--NOP	HALT	1W--NOP ENDS WITH WRONG RESULT
0412	DEC--NOP	HALT	1W--NOP ENDS WITH WRONG RESULT
0412	BLT--NOP	HALT	1W--NOP ENDS WITH WRONG RESULT
0412	CMPB--NOP	HALT	1W--NOP ENDS WITH WRONG RESULT
0412	BLE--NOP	HALT	1W--NOP ENDS WITH WRONG RESULT
0412	BEQ--NOP	HALT	1W--NOP ENDS WITH WRONG RESULT
0412	BR--NOP	HALT	1W--NOP ENDS WITH RIGHT RESULTS
0412	MOV--NOP	HALT	2W--NOP,1W ENDS WITH WRONG RESULT
0613	ADD--BIC	HALT	2W--2W,1W--1W ENDS WITH WRONG RESULT
0613	ADD--CMP	HALT	2W--2W,1W--1W ENDS WITH WRONG RESULT
0613	ADD--SUB	HALT	2W--2W,1W--1W ENDS WITH WRONG RESULT
0613	CMP--BR	TRAP	1W--1W UNUSED INSTRUCTION TRAP
0613	CMP--BIT	HALT	1W--1W ENDS WITH WRONG RESULT
0613	CMP--ADD	HALT	1W--1W ENDS WITH WRONG RESULT
0613	CMP--CMPB	HALT	2W--2W,1W--1W END SWITH WRONG RESULT
0613	DEC--INC	INFINITE LOOP	1W--1W

IT FINALLY ENDS BUT IT TAKES VERY LONG.

PROGRAM	FAULT	RESULT	COMMENTS
0613	DEC--TST	INFINITE LOOP	1W--1W NEVER ENDS
0613	DEC--COM	INFINITE LOOP	1W--1W
0613	DEC--JSR	TRAP	1W--1W UNUSED INSTRUCTION TRAP
0613	DEC--NONE	TRAP	1W--NONE UNUSED INSTRUCTION TRAP
0613	DEC--BNE	HALT	1W--1W WRONG RESULT
0613	DEC--MOV	HALT	1W--1W WRONG RESULT
0613	DEC--CMP	TRAP	1W--1W ODD ADDRESS TRAP CAN CHANGE TO INFINITE LOOP
0613	DEC--BIC	TRAP	1W--1W ODD ADDRESS TRAP CAN CHANGE TO EITHER WRONG RESULT OR INFINITE LOOP
0613	BR--BR	HALT	1W--1W ENDS WITH WRONG RESULT
0613	BR--BR	HALT	1W--1W ENDS WITH WRONG RESULT
0613	BR--NONE	TRAP	1W--1W UNUSED INSTRUCTION TRAP
0613	BR--BEQ	HALT	1W--1W ENDS WITH WRONG RESULT
0613	BR--BLT	HALT	1W--1W ENDS WITH WRONG RESULT
0613	BR--JSR	TRAP	1W--1W ODD ADDRESS TRAP
0613	BR--MOV	TRAP	1W--1W ODD ADDRESS TRAP

PROGRAM	FAULT	RESULT	COMMENTS
0613	BR--CMP	TRAP	1W--1W ODD ADDRESS TRAP
0613	BR--BIC	TRAP	1W--1W ODD ADDRESS TRAP
0613	BR--BMI	HALT	1W--1W ENDS WITH WRONG RESULT

(3) Functional Fault Injection in the Instruction Register

PROGRAM	FAULT	RESULT	COMMENTS
0613	I BIT 0 S-A-1	WAIT	
0613	I BIT 0 S-A-0	TRAP	ODD ADDRESS TRAP
0613	I BIT 1 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 1 S-A-1	INFINITE LOOP	
0613	I BIT 1 S-A-0	TRAP	RED STACK TRAP
0613	I BIT 2 S-A-1	TRAP	RED STACK TRAP
0613	I BIT 2 S-A-0	TRAP	RED STACK TRAP
0613	I BIT 3 S-A-1	TRAP	UNUSED INSTRUCTION TRAP
0613	I BIT 3 S-A-0	TRAP	ODD ADDRESS TRAP
0613	I BIT 4 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 4 S-A-0	TRAP	ODD ADDRESS TRAP
0613	I BIT 5 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 5 S-A-1	TRAP	RED STACK TRAP
0613	I BIT 5 S-A-0	TRAP	RED STACK TRAP
0613	I BIT 6 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 6 S-A-0	TRAP	ODD ADDRESS TRAP
0613	I BIT 7 S-A-1	INFINITE LOOP	
0613	I BIT 7 S-A-0	HALT	WRONG RESULT
0613	I BIT 8 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 8 S-A-0	TRAP	RED STACK TRAP
0613	I BIT 9 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 9 S-A-0	TRAP	UNUSED INSTRUCTION TRAP
0613	I BIT 10 S-A-1	INFINITE LOOP	
0613	I BIT 10 S-A-0	TRAP	UNUSED INSTRUCTION TRAP
0613	I BIT 11 S-A-1	TRAP	RED STACK TRAP
0613	I BIT 11 S-A-0	INFINITE LOOP	
0613	I BIT 12 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 12 S-A-0	WAIT	
0613	I BIT 13 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 13 S-A-0	TRAP	UNUSED INSTRUCTION TRAP
0613	I BIT 14 S-A-1	TRAP	ODD ADDRESS TRAP
0613	I BIT 14 S-A-0	HALT	ENDS WITH WRONG RESULT
0613	I BIT 15 S-A-1	TRAP	RED STACK TRAP
0613	I BIT 15 S-A-0	INFINITE LOOP	
0412	I BIT 0 S-A-1	WAIT	
0412	I BIT 0 S-A-0	HALT	WRONG RESULT
0412	I BIT 1 S-A-1	TRAP	ODD ADDRESS TRAP
0412	I BIT 1 S-A-0	TRAP	ODD ADDRESS TRAP
0412	I BIT 2 S-A-1	TRAP	UNUSED INSTRUCTION TRAP
0412	I BIT 2 S-A-0	TRAP	RED STACK TRAP
0412	I BIT 3 S-A-1	TRAP	RED STACK TRAP
0412	I BIT 3 S-A-0	TRAP	UNUSED INSTRUCTION TRAP
0412	I BIT 4 S-A-1	TRAP	UNUSED INSTRUCTION TRAP
0412	I BIT 4 S-A-0	INFINITE LOOP	
0412	I BIT 5 S-A-1	TRAP	UNUSED INSTRUCTION TRAP
0412	I BIT 5 S-A-0	HALT	END S WITH WRONG RESULT

PROGRAM	FAULT	RESULT	COMMENTS
0412	I BIT 12 S-A-1	TRAP	UNUSED INSTRUCTION TRAP
0412	I BIT 12 S-A-0	TRAP	ODD ADDRESS TRAP
0412	I BIT 13 S-A-1	TRAP	ODD ADDRESS TRAP
0412	I BIT 13 S-A-0	HALT	ENDS WITH WRONG RESULT
0412	I BIT 14 S-A-1	TRAP	ODD ADDRESS TRAP
0412	I BIT 14 S-A-0	HALT	ENDS WITH WRONG RESULT
0412	I BIT 15 S-A-1	TRAP	RED STACK TRAP

NOTE1: I REPRESENTS THE INSTRUCTION REGISTER
NOTE2: S-A-1(0) IS THE ABBREVIATION FOR STUCK-AT-1(0)

(4) Functional Fault Injection in the Data Manipulation Functional Module

PROGRAM	FAULT	RESULT	COMMENTS
0613	ALU⊕1	TRAP	ODD ADDRESS TRAP
0613	ALU+2	TRAP	RED STACK TRAP
0613	ALU+4	TRAP	RED STACK TRAP
0412	ALU⊕1	TRAP	ODD ADDRESS TRAP
0412	ALU⊕2	TRAP	RED STACK TRAP
0412	ALU⊕4	TRAP	RED STACK TRAP
0412	ALU⊕8	TRAP	RED STACK TRAP
0412	ALU⊕16	TRAP	RED STACK TRAP
0412	ALU+2	HALT	ENDS WITH WRONG RESULT
0412	ALU+4	TRAP	RED STACK TRAP
0412	ALU-2	TRAP	UNUSED INSTRUCTION TRAP
0613	ALU⊕4	TRAP	UNUSED INSTRUCTION TRAP
0613	ALU⊕2	INFINITE LOOP	EVER LASTING INFINITE LOOP OVER ONLY ONE INSTRUCTION
0613	ALU⊕3	TRAP	ODD ADDRESS TRAP
0613	ALU+1	TRAP	ODD ADDRESS TRAP