

DOCUMENT NO.  
PX 13243

LEVEL

SPERRY+UNIVAC

AD A099259

MIL-STD-1750 CERTIFICATION STUDY  
FINAL REPORT

CONTRACT NO. F33657-80-C-0118  
CDRL NO. 3

DTIC  
ELECTE  
MAY 21 1981  
C

PREPARED FOR

DTIC FILE COPY

USAF/AFSC  
Aeronautical Systems Division  
Mark for: ENAS/XRE/PMWB  
Wright-Patterson AFB, Ohio 45433

DISTRIBUTION STATEMENT A

Approved for public release;  
Distribution Unlimited

81 5 19 039

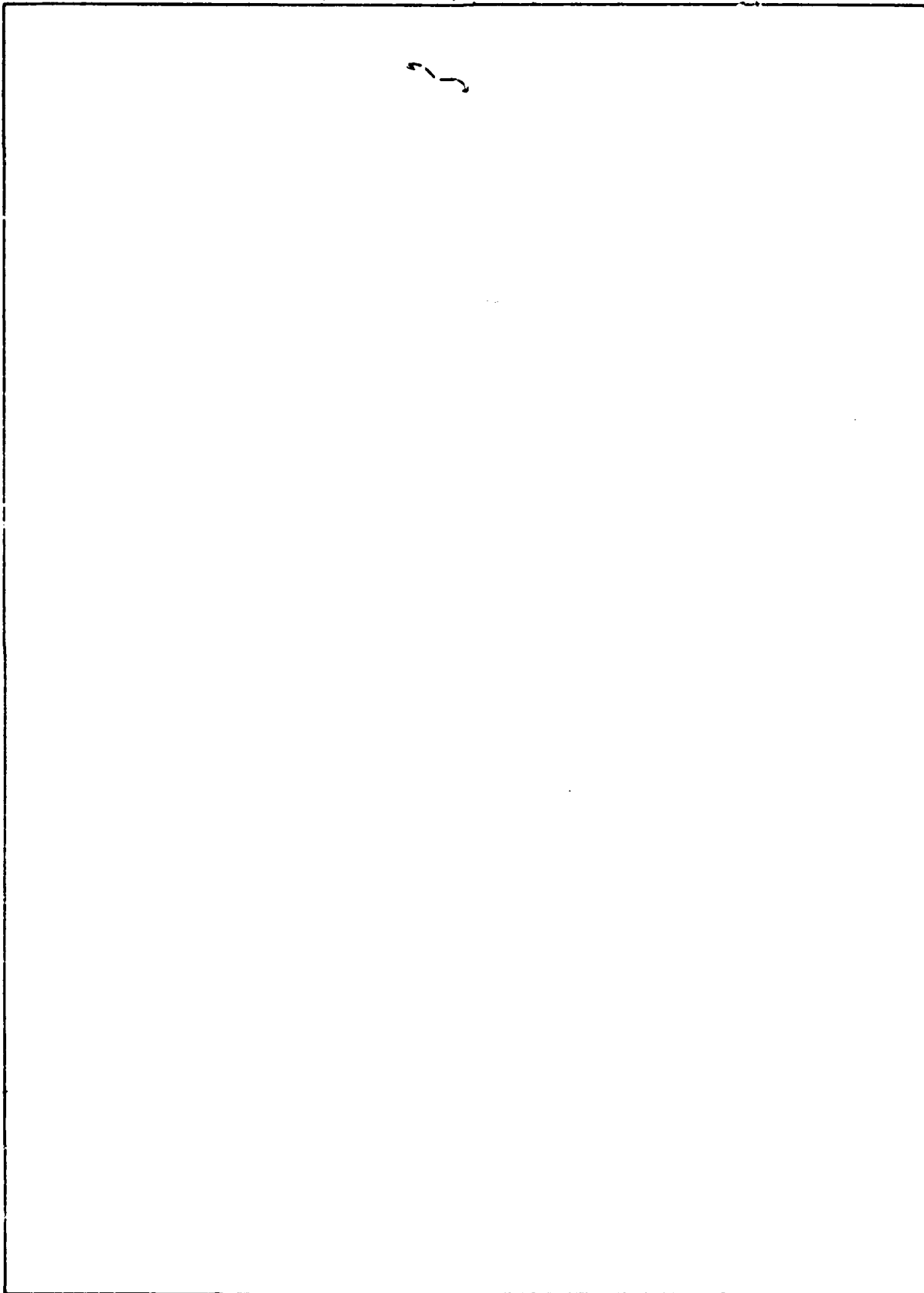
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
	AD-A099259	
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED	
(6) MIL-STD-1750 CERTIFICATION STUDY.	(9) FINAL REPORT. 21 Jan 1980 - 6 Jun 1980	
7. AUTHOR(s)	6. PERFORMING ORG. REPORT NUMBER	
	PX 13243	
	8. CONTRACT OR GRANT NUMBER(s)	
(14) UNIVAC-DSD PX-13243	(15) F33657-89-C-0118	
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Sperry Rand Corp. P.O. Box 3525 Sperry Univac Division St. Paul, MN Sperry Univac Defense Systems 55165		
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE
ASD/XRE WPAFB, OH 45433		6 Jun 1980
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		13. NUMBER OF PAGES
ASD/ENASD WPAFB, OH 45433		167
		15. SECURITY CLASS. (of this report)
		UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)		
<div style="border: 1px solid black; padding: 5px; text-align: center;"> <b>DISTRIBUTION STATEMENT A</b>            Approved for public release;            Distribution Unlimited         </div>		
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)		
Instruction Set Architecture MIL-STD-1750 Computer Architecture Verification Computer Testing		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		
This study evaluates a number of different validation techniques and procedures that can be used by the Air Force to validate a candidate computer for compliance with MIL-STD-1750.		

1

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)



SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE OF CONTENTS

1. INTRODUCTION AND SUMMARY
  - 1.1 Objective
  - 1.2 Approach
  - 1.3 Summary of Recommendations
2. BACKGROUND
  - 2.1 Reduction of Risk
  - 2.2 Hardware Independent Certification
  - 2.3 Testability Requirements
  - 2.4 Levels of Completeness
    - 2.4.1 Structural Assumptions
    - 2.4.2 Functional Assumptions
3. CERTIFICATION PROCEDURE ISSUES
  - 3.1 Resource Requirements
    - 3.1.1 Hardware Requirements
    - 3.1.2 Software Requirements
  - 3.2 A Secondary 1750 Golden Standard
    - 3.2.1 Golden Standard Uses
    - 3.2.2 Golden Standard Development
    - 3.2.3 Golden Standard Validation
  - 3.3 Certification Procedure Validation
    - 3.3.1 Completeness Validation
    - 3.3.2 Accuracy Validation
  - 3.4 Certification Procedure Control
    - 3.4.1 Control Functions
    - 3.4.2 Procedure Initialization
    - 3.4.3 Control Methods
  - 3.5 Certification Program Design
    - 3.5.1 Organization Factors
    - 3.5.2 Complexity Factors
    - 3.5.3 Coding Techniques

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	<i>Per 50 an file</i>
By	
Date	
<i>A</i>	



- 3.6 Certification Data Design
  - 3.6.1 Minimum Requirements for Operands
  - 3.6.2 Generation of Expected Results
  - 3.6.3 Storage of Operands and Expected Results
- 3.7 Certification Procedure Report Generation
  - 3.7.1 Methods of Generating Reports
  - 3.7.2 Content and Format of Certification Reports
  - 3.7.3 Detailed Failure Analysis

#### 4. CERTIFICATION PROCEDURE EVALUATION

- 4.1 Test Methodologies
- 4.2 Evaluation Criteria
  - 4.2.1 Efficiency
  - 4.2.2 Reliability
  - 4.2.3 Cost
- 4.3 Application of the System Efficiency Model
  - 4.3.1 Simplifying Observations
  - 4.3.2 Implications of Model as Design Criteria
- 4.4 The Role of System Reliability
  - 4.4.1 Expanded Definition of Completeness
  - 4.4.2 Expanded Definition of Clarity
  - 4.4.3 Expanded Definition of Verifiability
- 4.5 Certification Procedure Costs
- 4.6 Design Goals
- 4.7 Evaluation of Candidate Methodologies
  - 4.7.1 Description of Candidate Methodologies
  - 4.7.2 Detailed Evaluation
    - 4.7.2.1 Evaluation of Test Completeness
    - 4.7.2.2 Evaluation of Context Testing
    - 4.7.2.3 Evaluation of Ease of Use
    - 4.7.2.4 Evaluation of Test Expandability
    - 4.7.2.5 Evaluation of Test Validity
    - 4.7.2.6 Evaluation of Test Adaptability
    - 4.7.2.7 Evaluation of Test Maintainability
    - 4.7.2.8 Evaluation of Test Costs

#### 4.7.3 Summary of Candidate Methodologies

### 5. RECOMMENDED APPROACH

- 5.1 Overview of Recommended Approach
- 5.2 Test Control Program
- 5.3 Protocol Handler
  - 5.3.1 VAX Resident Protocol Handler
  - 5.3.2 UUT Resident Protocol Handler
- 5.4 Data Link Drivers
  - 5.4.1 RS-232C Data Link Drivers
  - 5.4.2 MIL-STD-1750 Data Link Drivers
- 5.5 MIL-STD-1750 Simulator
- 5.6 MIL-STD-1750 Test Programs
  - 5.6.1 Factory Acceptance Tests
  - 5.6.2 Data Link Test
  - 5.6.3 Instruction Tests
  - 5.6.4 Memory Test
  - 5.6.5 Register Tests
  - 5.6.6 Derived Address Tests
  - 5.6.7 Input/Output and Interrupt Tests
  - 5.6.8 Jump and Branch Tests
  - 5.6.9 Random Instruction Sequence Tests
  - 5.6.10 Context Tests
- 5.7 Test Data Generation
  - 5.7.1 Test Data Generation Using the Cross Assembler
  - 5.7.2 Test Data Generation Programs
  - 5.7.3 Test Data Generation Using the 1750 Simulator
- 5.8 Test Validation
  - 5.8.1 Design and Validation of the Simulator and the Test Program
  - 5.8.2 Data Generation for the Initial Test Program
  - 5.8.3 Independent Verification of the Test Validation
- 5.9 Reporting Test Results

- 5.10 Changes to MIL-STD-1750
- 5.11 Distribution and Control of Certification Tests
- 5.12 System Resource Requirements
- 5.13 Summary of Resources to be Developed

6. REFERENCES

- APPENDIX A. Optional Aspects of MIL-STD-1750
- APPENDIX B. Recommendation for Changes to MIL-STD-1750
- APPENDIX C. MIL-STD-1750 Simulator Subroutines
- APPENDIX D. Protocol Handler Subroutines
- APPENDIX E. Sample Assembly Code for Generating Tables  
of Operands and Results
- APPENDIX F. Review of the AFAL (DAIS) AN/AYK-15A ATP  
Program Design
- APPENDIX G. Review of the Sperry Univac AN/AYK-15A  
Acceptance Test Program Design
- APPENDIX H. Sorted List of MIL-STD-1750 Instructions

LIST OF FIGURES	<u>AFTER PAGE</u>
3-1. Majority Vote Validation	3-8
3-2. Use of Assembly Time Operations to Generate Expected Results . . . . .	3-24
3-3. Sample Page from AFAL AN/AYK-15A ATP Report . . .	3-27
3-4. Instruction Code Test -- Detailed Failure Report .	3-28
3-5. Instruction Code Test -- Summary Failure Report .	3-28
4-1. System Efficiency Model . . . . .	4-4
4-2. Architectural Entities that Require Thorough Testing	4-14
4-3. Relationship Between Risk and Cost . . . . .	4-16
4-4. Possible Cost Curves for Four Certification Options	4-18
4-5. Rank Order Evaluation of Test Methodologies . . .	4-22
5-1. MIL-STD-1750 Certification Facility . . . . .	5-1
5-2. Instruction Test Control Structure . . . . .	5-9
5-3. Typical Loop Test (D, DX Format Integer Instruction)	5-9
5-4. Register Test Program . . . . .	5-11
5-5. Addressing Modes in the 1750 Instruction Set . .	5-12
5-6. Sample of Code for Jump test . . . . .	5-14
5-7. Conditional Branch/Jump Matrix . . . . .	5-14
5-8. Certification System Resources . . . . .	5-26
5-9. Test Resources to be Developed for ISA Certification	5-27

## 1. INTRODUCTION

Current Air Force Avionics systems have a multiplicity of computer architectures, system interface technologies, and related software systems resulting in high development, acquisition, and life-cycle costs. As a means of reducing these costs and simplifying systems development, MIL-STD-1750 has been established as the Air Force standard instruction set for Avionics computer applications. Several efforts are under way to implement MIL-STD-1750 for test and evaluation in avionics systems. The ASD/ENA Systems Engineering Avionics Facility (SEAFAC) has been given the responsibility for certifying compliance of vendor produced computers with MIL-STD-1750. This feasibility study of certification procedures was initiated to support development of a certification facility at SEAFAC. The result of this study is a discussion of background and procedural issues of instruction set architecture certification; an evaluation of test methodologies; and a set of recommendations for a MIL-STD-1750 certification procedure. The report is relevant to the large class of computers that are defined in terms of an instruction set architecture (ISA) which may be implemented in a variety of hardware architectures.

### 1.1 Objective

The objective of this study is to identify, evaluate, and select from a wide variety of validation techniques and procedures that can be applied by SEAFAC to validate a candidate computer for compliance with MIL-STD-1750. The resulting certification procedure is a means by which SEAFAC can verify that a unit under test (UUT) behaves as required by MIL-STD-1750, regardless of the specific technology employed to implement the design or of the end use for which the UUT was intended. The only characteristics that various implementations can be expected to have in common are those stated in MIL-STD-1750.

Since these requirements are stated in the form of computational processes, it follows that compliance can be verified by some form of computational testing.

The fundamental assumption behind and major reason for developing a certification procedure is that validating the compliance of a particular 1750 implementation will reduce the risk that the hardware does not conform with MIL-STD-1750. It is assumed, in addition, that the more thorough or complete the testing is, the lower the risk will be. Because of the large number of possible states of the 1750 processor, registers, and memory, it is possible that a complete test of any 1750 computer would take many years. Therefore, a certification procedure is viewed as making reasonable tradeoffs between completeness and other evaluation criteria such as cost and efficiency.

Consistent with the understanding that MIL-STD-1750 is a specification of an instruction set architecture and is independent of particular hardware considerations, it is assumed that the certification procedure requires no knowledge of the particular hardware architecture of the UUT. This assumption requires that to whatever extent possible, the UUT be treated as a black box, and it implies that the test may not make use of hardware implementation details that might normally be used to simplify testing procedures.

This understanding also means that a test based upon a structural analysis of the UUT is ruled out, as is any test procedure which involves inserting hardware probes into the UUT. Instead, the test procedure is assumed to be based upon a functional analysis of MIL-STD-1750. The goal of the testing procedure is to verify that the UUT behaves as required by ascertaining that it executes properly. It does that by detecting and

reporting any functional flaws that might be in the UUT. The specifics of all test failures, including the circumstances and conditions required to reproduce the failure are of interest, while the ability to detect and identify hardware component failures is not of concern.

Within some constraints of reasonable test program complexity and execution time, the certification procedure should be as complete as possible in verifying the functional characteristics of the 1750 implementation under test. The motivation for pursuing such completeness comes primarily from the realization that the certification procedure requires no prior knowledge of the hardware used to implement MIL-STD-1750. Arguments about related functions or instruction codes using the same hardware and thus not requiring separate tests are not theoretically justified. In practice, however, when exhaustive testing is not practical, certain assumptions about hardware structure can serve as guidelines for reducing the magnitude of the test problem. These structural assumptions lead to a function oriented view of the 1750 instruction set architecture.

## 1.2 Approach

The approach to this study of certification procedures involved three phases: analysis of test methodology components, identification and application of evaluation criteria, and recommendation of specific procedures, test designs, and support tools for use in certification. The first phase of the study involved reviewing the literature related to ISA testing, organizing knowledge about design certification available through the experience of people at Sperry Univac, and examining the two available Acceptance Test Programs for the AN/AYK-15A (the AFAL (DAIS) ATP, SA 421 206 and the Sperry Univac Confidence Test). A brief review of these two methodologies is

presented in Appendices F and G. All of the information gathered was well mixed with ideas from the authors in producing the contents of the report. Key articles from the literature are referenced in Section 6.

The analysis of test methodology components is presented in Section 3 as a discussion of certification procedure issues. Section 3.1 offers a discussion about hardware and software resources not implicit in MIL-STD-1750 that might be required for certification of a 1750 device.

Section 3.2 explores the concept of a "golden standard", as a hardware or software device which implements MIL-STD-1750 and serves as a functioning reference tool which could be used in development, operation, and validation of certification tools and procedures. Section 3.3 then underlines the importance of validating whatever certification procedure is developed, and it discusses some dimensions of that problem. The rest of Section 3 provides some details about the major components of test methodologies, namely, certification procedure control, program and data design, and report generation. Individually the subsections introduce approaches to control, program design and data design which are motivated by a belief in the benefits of a well-structured procedure. These benefits include ease of use, validation, understanding, and modification. The final section identifies a range of report generation techniques that might be used in a selected test methodology.

The identification and application of evaluation criteria is the subject of Section 4. Section 4.1 defines what a test methodology is in terms of its attributes, and discusses the methods of combining alternatives to each of these attributes. Criteria for evaluating test methodologies are discussed in



Section 4.2 and a system efficiency model is developed in Section 4.3. Sections 4.4 and 4.5 expand upon previously introduced evaluation criteria until certification procedure design goals are introduced in Section 4.6. Finally, an evaluation of currently known certification procedure options available to SEAFAC are introduced in Section 4.7. The overall effect of the discussion of test methodologies and evaluation procedures is to define relations between factors such as risk, cost, reliability, and completeness. The system efficiency model is used generatively to develop design goals that are achievable and cost effective.

### 1.3 Summary of Recommendations

The certification test methodology recommended by Sperry Univac consists of procedures for loading, running, and reporting on an extensive set of certification test programs designed to operate in the MIL-STD-1750 unit under test. These programs are designed to maximize the number of instruction codes and instruction code sequences to be tested. An automated method of generating test data is provided. This automated test data generation capability allows expansion of test data to a level of completeness limited only by time and/or cost constraints.

A MIL-STD-1750 instruction set simulator is recommended to provide one of the independent methods of generating test data and to serve in the validation of the certification test programs and data for correctness and accuracy. It is recommended that all test programs and automated test data generation mechanisms undergo rigorous validation procedures, including self-test and validation against existing software components, before they are utilized for certification of any implementations of MIL-STD-1750.

It is further recommended that the VAX-11/780 computer system currently installed at SEAFAC host the program generation, storage, and maintenance capabilities needed to support efficient and effective MIL-STD-1750 certification procedures. The use of available data links is specified to allow direct connection to the VAX-11/780 when appropriate hardware capabilities are incorporated into the MIL-STD-1750 implementation under test. This direct connection allows fully automatic operation of all test procedures and assures proper operation and application of test procedures by minimizing human intervention. Summary reports produced by a VAX-11/780 resident test control program provide descriptive information about specific options that were tested. Further details are contained in Section 5.

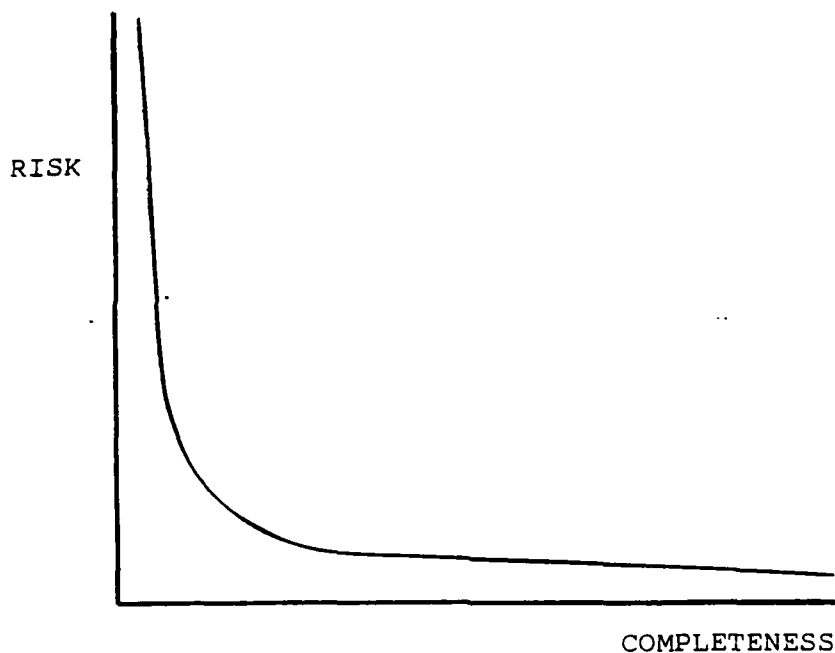
## 2. BACKGROUND

Compliance of vendor produced computers to MIL-STD-1750 is of major importance in the Air Force program to reduce life-cycle costs. Fostering software commonality through standardization of an instruction set architecture can only be realized if functional specifications are specific, testable, and uniformly applied. To this end, the MIL-STD-1750 control board and the MIL-STD-1750 Users Group are in the process of revising the standard to improve definition and clarity and to extend functional capabilities. SEAFAC, with the help of industry representatives, is preparing to develop the necessary techniques, tools, and experience to perform a thorough design certification procedure. The goal of this procedure is to verify that the UUT behaves as required by MIL-STD-1750. As a basis for discussion of what the certification procedure issues are, some assumptions about the process of certifying a 1750 computer are offered here.

### 2.1 Reduction of Risk

The MIL-STD-1750 Instruction Set Architecture (ISA) allows multiple vendors to compete for a particular avionics computer application while eliminating architecture proliferation in Air Force avionics systems. Vendors are allowed complete discretion in the detailed design of an avionic computer within the physical and performance requirements for the application. This wide latitude of vendor design and implementation requires that the characteristics of the implementation conform exactly in functional specifications rather than in construction details. The purpose of certification is to verify that the functional characteristics of candidate Air Force avionic computers comply with MIL-STD-1750.

The fundamental assumption, and in fact the major reason for the certification procedure, is that by validating the design of a particular implementation, the risk of that computer not complying with MIL-STD-1750 will be reduced. It is assumed, in addition, that the more thorough or complete the certification testing is, the lower the risk will be. This is shown graphically by the following curve, which is asymptotic to both the risk and completeness axes. Risk is at a maximum when no testing is done.



and approaches zero as the amount of testing increases. This curve also indicates that with relatively little testing the risk can be substantially reduced, and that exhaustive testing will yield only limited further reductions in risk.

## 2.2 Hardware Independent Certification

Vendor provided tests of conformance to specifications are normally organized in a manner that will test and diagnose the largest number of possible error conditions with the minimum amount of test code. This is usually done by utilizing specific knowledge of the processor design to reduce the number of tests conducted. For example, a particular 1750 implementation could use the same adder logic for incrementing the instruction counter, processing ADD instructions, and indexing; therefore, only tests of the ADD instruction might be used by the vendor to validate operation of this one adder. This vendor provided test might be perfectly adequate to test the specific implementation described but it would not be adequate to test another implementation that utilized pipeline techniques and multiple adders to achieve greater throughput.

Consistent with the understanding that MIL-STD-1750 is a specification of an instruction set architecture and is independent of particular hardware considerations, it is assumed that the certification procedure requires no knowledge of the particular hardware architecture of the UUT. This assumption requires that to whatever extent possible, the UUT be treated as a black box, and it implies that the test may not make use of hardware implementation details that might normally be used to simplify testing procedures.

This understanding also means that a test based upon a structural analysis of the UUT is ruled out, as is any test procedure which involves inserting hardware probes into the UUT. Instead,

the test procedure is assumed to be based upon a functional analysis of MIL-STD-1750. The goal of the testing procedure is to verify that the UUT behaves as required by ascertaining whether it executes properly. It does that by detecting and reporting any functional flaws that might be in the UUT. The specifics of all test failures, including the circumstances and conditions required to reproduce the failure are of interest, while the ability to detect and identify hardware component failures is not of concern.

### 2.3 Testability Requirements

Because the standard is stated in terms of computational processes, it is assumed that certification will require one or more test programs to be generated for execution on the UUT. These programs should, at a minimum, exercise and verify each functional capability specified in the standard. The results of the interactions between the test program and the UUT will be used to determine compliance with MIL-STD-1750. Results will be considered acceptable if they do not conflict with information in the standard.

In order to run test programs it is assumed that certification of computer compliance with the standard is directed towards an implementation which is a general purpose programmable computer. In particular, the UUT is assumed to be able to run a program that utilizes a specific subset of non-optional instructions and requires some minimum amount of memory to be implemented. This statement is intended to exclude from consideration any implementations of MIL-STD-1750 which are severely limited subsets of the standard in either instruction repertoire or in memory.

Another testability assumption is that mechanisms for loading programs and examining results are available. This assumption is important because MIL-STD-1750 does not address questions of electrical interface to the ISA. Instead, the functional characteristics are specified, and any system constraints such as physical and environmental characteristics are left to the system designer. Program loading, program execution and status monitoring capabilities can be designed to meet specific mission requirements.

Isolation of system constraints is a distinct advantage to the system designer, but presents a unique challenge to the certification test procedure. The extreme flexibility required to test computers that could range from ultra small integrated components, to relatively large stand-alone versions, to computers completely embedded in higher level system components is a significant challenge. The ability to completely specify internal connections to a compliant computer would make testing and validation much easier, but it might also unnecessarily constrain the technology that could be used for implementation.

#### 2.4 Levels of Completeness

Within some constraints of reasonable test program complexity and execution time, the certification procedure should be as complete as possible in verifying the functional characteristics of the implementation under test. The motivation for pursuing such completeness comes primarily from the realization that the certification procedure requires no prior knowledge of the hardware used to implement MIL-STD-1750. Arguments about related functions or instruction codes using the same hardware and thus not requiring separate tests are not theoretically

justified. In practice, however, when thorough testing is not possible, certain assumptions about hardware structure may be acceptable as guidelines for reducing the magnitude of the test problem. These structural assumptions lead to a function oriented view of the 1750 instruction set architecture.

#### 2.4.1 Structural Assumptions

One such assumption about the 1750 functional structure is that derived address calculations use common circuits for all instructions of the same address mode. Thus, a complete test of address calculation for one OPCODE need not be repeated for others of the same address mode. However, because a particular OPCODE may not be implemented or decoded properly, each separate instruction should still be tested in a limited way for its ability to use the common addressing mode hardware.

The premise here concerning the interaction between separate instructions and common addressing modes is that it is unreasonable to completely test the derived address calculation for each instruction. There are of course cases where for even one particular instruction it is not reasonable to exhaustively test its function. A specific example is the testing of arithmetic instructions for correct operation over all values of both operands. An adequate, though limited, level of testing for these instructions would depend on a careful selection of data. This is discussed in detail in Section 3.6. So the question remains for each situation where complete testing is unreasonable as to what level of testing is adequate. One answer is to make assumptions about how the 1750 architecture would be implemented; then exploit the assumed structure to reduce the test complexity, while maximizing the effec-



tive level of completeness. Another answer for codes which perform arithmetic and logical functions on data is to do a careful job of selecting test data. Such a selection process might well include the random generation of data as a means of sampling the space of all possible values.

#### 2.4.2 Functional Assumptions

Pursuing the assumption that derived address calculations use a common hardware path for all instructions leads to the observation that the storage locations referenced by the address calculation -- that is, the registers and memory -- might reasonably be tested separately from tests for the individual OPCODES. If we also accept the additional assumption that verification of the ability of each register and memory cell to hold all data values can be done once instead of separately for each OPCODE, then any certification of 1750 functions can be divided into tests of registers, memory, and instruction codes.

The register and memory tests would focus on confirming the ability to address all locations and write and read all possible data values in those locations. In the case of memory, it may not be reasonable to try all bit patterns in every location, so some sort of compromise in the level of completeness may be required. Some bit patterns commonly used in memory tests include all ones, all zeros, shifting ones, shifting zeros, alternating ones, varying length sequences of ones/zeros, and addresses.

With confirmation that derived addressing and that arbitrary data value storage function properly, then the individual OPCODE tests are free to pursue verifying the correct performance of the operation aspect of the instruction. This verification involves trying all data values necessary to reasonably confirm the function of the instruction, particularly with arithmetic and logical operations. There is still the need for each instruction to confirm its ability to use the addressing structure previously confirmed as a general capability. That is, for each OPCODE test, a variety of registers and derived addresses should be employed. One method of selecting registers to use that would ensure a good variety of register use is a random selection process embedded in the test program creation procedure. For OPCODES that allow use of an index register, it would be important to try the OPCODE with  $RX = R0$  as well as with a selection of actual registers (i.e., try the instruction with and without indexing).

The instruction code tests, then, verify that with specified operands, the correct result is produced. The result includes not only the specific answer produced by the instruction -- e.g., the sum resulting from the ADD instruction -- but also the correct setting of the status word and the correct generation of interrupts as appropriate -- i.e., fixed and floating point overflow, floating point underflow, or illegal OPCODE interrupts. To be complete, the notion of correct setting would include not only that the status word is set and/or interrupts occur when required, but also that they do not get set or occur when they are not supposed to.

Unfortunately, the concept of confirming that undefined changes do not occur represents a task which is at least an order of

magnitude more difficult than verifying the specific correct actions of each instruction. For example, in doing the memory test, the required result of storing a data value at a particular location is that the data value correctly reach the intended location, an action verified by reading the value from that location. But to verify that no unintended action occurs implies checking all other locations in memory to determine whether the store operation sent the data value elsewhere in error.

An appropriate compromise in level of completeness might be to check for unintended actions on the basis of instruction function. Thus, a check that the word being transferred goes only to the correct place would only be done with loads and stores, and not arithmetic instructions. Checks for overflow/underflow interrupts would always be done with arithmetic instructions, whether the data is supposed to produce them or not, but such a check would not be done for loads and stores. The full complement of locations which might possibly be checked includes the general registers, special registers, and all of memory.

So far, the discussion has identified that registers, memory, and individual OPCODES are the major components of the 1750 to test for correct operation. These components cover most 1750 functions. Including the input and output commands and the full complement of possible interrupts as separate though overlapping categories provides a more complete picture of the architecture. Although the individual OPCODES verify the functioning of interrupts for overflow, underflow, and illegal instructions, the interrupt structure needs to be specifically verified with respect to the operation and interaction of the interrupt mask (IM), pending interrupt (PI), and fault (FT)

In summary, a relatively complete procedure for certifying compliance with MIL-STD-1750 needs to examine the areas of register and memory function, specifically addressing modes and data storage; the operation of all OPCODES, including a check of all illegal codes; the full operation of the interrupt mechanisms; and the capabilities for I/O commands not verified elsewhere in the separate tests.

### 3. CERTIFICATION PROCEDURE ISSUES

The purpose of this section of the report is to raise and discuss the key issues involved in specifying certification procedure test methodologies. Section 3.1 offers a discussion about hardware and software resources not implicit in MIL-STD-1750 that might be required for certification of a 1750 device. Section 3.2 explores the concept of a "golden standard", as a hardware or software device which implements MIL-STD-1750 and serves as a functioning reference tool which could be used in development, operation, and validation of certification tools and procedures. Section 3.3 then underlines the importance of validating whatever certification procedure is developed, and it discusses some dimensions of that problem. The rest of Section 3 provides some details about the major components of test methodologies, namely, certification procedure control, program and data design, and report generation. Individually, the subsections introduce approaches to control, program design and data design which are motivated by a belief in the benefits of a well-structured procedure. These benefits include ease of use, validation, understanding, and modification. The final section identifies a range of report generation techniques that might be used in a selected test methodology.

#### 3.1 Resource Requirements

The assumptions made in Section 2 indicate that at a minimum, any certification procedure would consist of test code which can be loaded and executed on a 1750 implementation capable of executing programs of moderate size. As a matter of practical necessity, these assumptions impose some requirements for hardware and software resources which are beyond those contained in

MIL-STD-1750. This section will discuss some minimum requirements in these areas and attempt to identify a range of resources needed to support the certification procedures mentioned in the rest of Section 3.

### 3.1.1 Hardware Resources

Hardware resources of the UUT necessary to support a certification procedure include a loading mechanism, control functions (processor reset, start, and stop), and a processor state display. For a certification procedure which is self-contained in the UUT, these requirements could translate into a control panel with switches that implement the control functions, and lights that provide the display capability; a bootstrap loader in ROM or entered through the switches; and an input device such as magnetic tape for loading the certification programs. For a certification procedure in which the test program is loaded and monitored from a test control computer, the hardware needed would be the same as before plus the test control computer and a communication channel to the UUT. The previous resources would still be needed to load communication software into the UUT. A variant of the procedure utilizing a standard interface between the two computers involves enhancing the channel to be a special purpose data and control interface which allows the loading, control, and display functions to be performed directly by the test control computer. This special hardware interface could replace the locally defined control panel and bootstrap loader.

A separate hardware resource that might be required by some certification procedures is a MIL-STD-1750 reference implementation or golden standard. Its previously certified, known good performance could be used in the development, execution, or validation of the certification procedure, as detailed in

## Section 3.2.

### 3.1.2 Software Resources

In addition to the certification test programs themselves, certain software resources might be required to support various certification procedures. Already mentioned was the need for communication protocol software by procedures using a channel for interaction between the UUT and a test control computer. But prior to that stage of the certification process, a 1750 Assembler or Cross Assembler would probably be necessary for the development of the test programs. The development and general administration (control, maintenance, and modification) of the programs could reasonably require file manipulation software such as is generally available on large time-shared computers.

During the development, execution, or validation of the certification procedure, the need for a golden standard as mentioned above could be met by a software implementation of MIL-STD-1750. Such a simulator could conceivably run on 1750 hardware, but the creation and use of 1750 simulator software would more reasonably be pursued on a test control computer, as outlined in Section 3.2. The generation of test data used in certification test programs might require a partial 1750 simulation, such as for a 1750 arithmetic and logical operation simulator. This idea is expanded in Section 3.6.

### 3.2 A Secondary 1750 Golden Standard

At several stages in the development of the certification procedure, the potential need arises for programs to have direct computational access to a 1750 equivalent computer, either in the form of a hardware device or a software simulation. Such a 1750 reference or "golden standard" is secondary to the primary MIL-STD-1750, but serves as a functioning standard for determining arithmetic, logical, or other machine state results from the execution of any desired sequence of 1750 instructions. The purpose of this section is to introduce the concept of a golden standard, outline the possible uses of the standard, and propose how a standard might be developed and validated.

#### 3.2.1 Golden Standard Uses

One important potential use of a golden standard 1750 would be to generate or validate the test data results used by the certification procedure. For example, a programmer might well use an ordinary desk calculator to determine arithmetic results for 16 bit integer operations. In this case, the calculator serves as a golden standard for a particular 1750 function. It would probably not be adequate, however, for producing arithmetic results for extended floating point calculations. Such data might be successfully produced manually, relying upon hand calculations and the programmer's ability to interpret correctly the definitions in MIL-STD-1750. But in practice, a specific golden standard 1750 device would be very useful for calculating floating point or any other test data results. A golden standard for 1750 arithmetic could be built into an assembler which had the ability to evaluate expressions and thereby calculate test results stored in the program at assembly time. But even if the original test data results were calculated



manually, it would be of value to employ a golden standard for later independent validation of the numbers.

Another potential use of a golden standard in the certification procedure is to execute programs in parallel with the running of those programs in the UUT. Results obtained from the two 1750 devices could be compared and any discrepancies attributed to errors in the UUT. One example of this use is to compare the operation of a particular instruction on randomly generated operands to see if the UUT produces the correct result. In fact, having a golden standard is the only practical way randomly generated data can be employed in the certification, because it is the only run-time method of obtaining a known good result.

A third use that a golden standard might serve is to aid in the process of validating the certification procedure itself. The approach is to simply run the certification procedure on the golden standard as if it were the UUT. Any procedure that runs without error is necessarily an accurate procedure since a golden standard is by definition error free.

### 3.2.2 Golden Standard Development

The development of a golden standard MIL-STD-1750 implementation is analagous to the certification by the National Bureau of Standards (NBS) of a secondary standard meter. The NBS certification implies that the secondary standard is a "true" meter with traceability to the "standard" meter. The big difference in the development of the golden standard MIL-STD-1750 implementation is that the "true" or first level standard exists on paper only, while the secondary standard is either hardware or a software simulation.

A hardware golden standard is of course what every 1750 implementation purports to be, and the existence of such a computer is the reason for creating a certification procedure. So if the decision were to develop a hardware 1750 golden standard, then the selection of any actual implementation would do as well as any other. A major concern with the use of a hardware golden standard is the susceptibility of components to fail over time, with the resulting necessity of re-certifying the computer periodically.

The other possibility for developing a golden standard is a software (or firmware) simulation (or emulation) of MIL-STD-1750. This approach shares a similar problem to the hardware implementation: namely, going from MIL-STD-1750, which is on paper, to the secondary standard involves interpretation and translation of the primary standard, with the accompanying possibilities for error. However, there are a number of important advantages to a simulation in addition to that of independence from component failures. Because a simulator serves, as a hardware implementation does, to translate a paper definition into a working model, it provides an opportunity to resolve any ambiguities or weak areas in the standard. It also provides an opportunity to examine the potential impact of any changes in the instruction set architecture without actually making any hardware changes.

A straightforward approach to developing a 1750 software golden standard would be to work directly from the definition of MIL-STD-1750 using an appropriate high-level language such as FORTRAN, together with the system software resources of the host computer. A 1750 emulation could be accomplished with the aid of special purpose hardware such as the Nanodata QM-1, as described by Clark and Troutman (1979). Or a golden standard

could be developed with the use of a register transfer language or computer hardware description language (CHDL). This approach, which is already utilized by MIL-STD-1750 to describe aspects of individual instructions, has been successfully pursued by a multitude of CHDL's (Shiva, 1979).

The motivating factor for considering the use of a CHDL in describing 1750 is that the major CHDL's have well-developed simulators for their language; thus, a 1750 CHDL description simulation would be the 1750 golden standard. The value of a CHDL description of 1750 would be not only to provide this simulation capability, but also to produce a compact, formal description of 1750 which lends itself to automatic design verification at various levels of detail. For example, languages such as SMITE and ISP provide syntax rules whose application to the CHDL description of 1750 would allow checking for any ambiguities.

To insure that an extension of MIL-STD-1750 into a CHDL would be successful and that all implementers of 1750 were working from the same description, it would be appropriate for the CHDL description of 1750 to become the true binding standard, with the existing written descriptions becoming non-binding explanations. Use of a CHDL for computer architecture description has been accomplished successfully for a number of existing computers (Barbacci et.al., 1977) and would enjoy the benefits of considerable government sponsored research and development work in the area of computer architecture specification, evaluation, and validation (Barbacci et.al, 1979, and Advanced SMITE Training Manual, 1979).

### 3.2.3 Golden Standard Validation

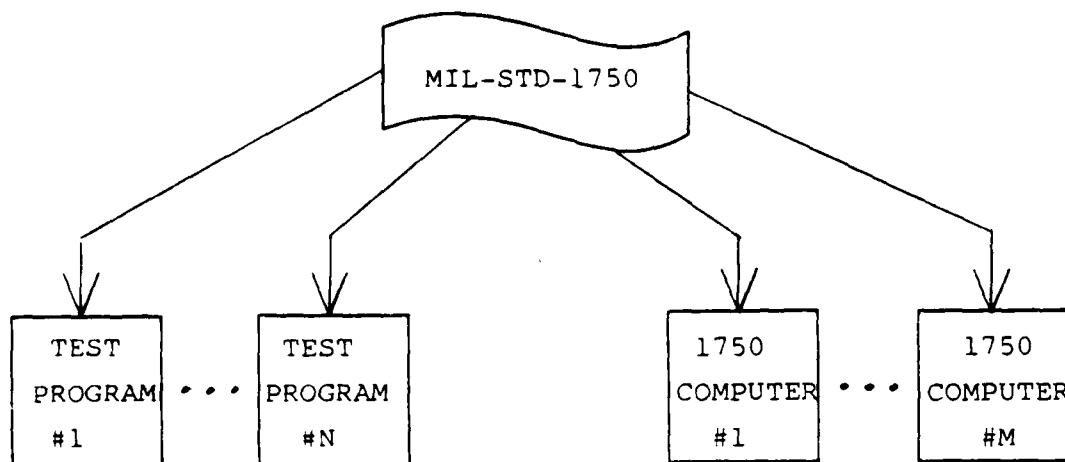
Use of a golden standard 1750 requires validation of that standard. This is in essence a chicken-and-egg problem which

requires certifying the golden standard machine for use in the certification procedure. There are three fundamental methods of validating a golden standard. "Self Test" methods of validation include manual test and analysis of the proposed standard, use of building block tests of individual functional (or structural) components, and/or algorithmic test and functional capabilities (e.g., checking multiplication by repeated addition). "Majority-vote" methods of validation require several copies of independent test programs to be run on one or more independently produced 1750 implementations. Each of the tests should be thorough tests of all 1750 functions and must be generated independently of the proposed standard implementation (s) and other test procedures as shown in Figure 3-1. The third method of validating a standard is by defining a particular implementation as being the golden standard. This third method has the disadvantage of diminishing the roll of the MIL-STD-1750 document to the role of a design document and makes the particular implementation the formal standard.

### 3.3 Certification Procedure Validation

An important issue in the overall design of a certification procedure is the consideration of how to validate its completeness and accuracy. The validation of completeness involves establishing to a high degree of certainty that the certification procedure tests for full compliance with MIL-STD-1750, leaving no required part of the standard untested. The validation of accuracy is a matter of developing confidence that the certification procedure contains correct reference results in its various tests of individual MIL-STD-1750 functions. In other words, whenever the procedure is applied to a MIL-STD-1750 implementation, it is essential that reported errors do in fact represent problems in the UUT and not problems in the procedure. In practice, it is likely that validation of the certification procedure will be an on-going process which

FIGURE 3.1. MAJORITY VOTE VALIDATION



- Run all N programs on all M implementations of MIL-STD-1750.
- Resolve all conflicts by reference back to MIL-STD-1750
  - The computer may have a flaw.
  - The program may have a bug.
  - The MIL-STD may be ambiguous.
- Results of this method improve as
  - The number of independently produced test programs increase
  - The number of independently produced 1750 implementations (including simulators) increases, and
  - The completeness of each test program increases.

asymptotically approaches completion; any errors detected in a UUT will require careful examination to confirm that the error is not in the procedure or in its interpretation of an ambiguous aspect of the standard.

### 3.3.1 Completeness Validation

Validation of the completeness of the certification procedure is initially accomplished with a careful design process that matches a test procedure with each function of the MIL-STD-1750. Part of this process is a clear understanding of which aspects of MIL-STD-1750 are optional and which are required, and which aspects are required but not well enough defined to be tested. Aspects that have been noted in these categories are listed in Appendices A and B. Recommendations for changes to more precisely define MIL-STD-1750 have been made to the MIL-STD-1750 Control Board by the MIL-STD-1750 Users Group. These recommendations are included within MIL-STD-1750A.

Functional capabilities required by MIL-STD-1750 can be uniformly and unambiguously tested for all candidate implementations; optional features can be tested for their existence in the UUT (features not implemented should generate illegal OPCODE interrupts) and extensive tests can be automatically initiated for those optional features that are functionally specified by the standard. Separate test procedures can be utilized to test those optional features that are allowed by MIL-STD-1750 but are fully specified only within documents associated with a specific implementation of MIL-STD-1750.

A procedure which has been used successfully in the area of software testing to assess the completeness of test programs is to introduce errors in the object being tested and determine if the test program detects them (Budd et.al, 1978). Since

the certification procedure is actually the entity being tested, the 1750 implementation should be a reference or golden standard 1750 so that errors are introduced in a controlled way to a known good implementation. Since introducing errors requires potentially complex modifications to the golden standard implementation, it is more desirable to use a software simulation golden standard.

### 3.3.2 Accuracy Validation

In addition to validating the completeness of the certification procedure, it is important to consider methods of validating its accuracy. One approach is to use the computer itself to calculate test answers using alternate algorithms. For example, verify the accuracy of multiplication test answers by repeated addition of the operand to form the product. Similarly, division can be verified by repeated subtraction, subtraction by addition of the two's complement, addition by a series of logical operations, and so on, in a hierarchical ordering that assumes the correctness of levels below it. At the lowest level, some manual checking or exhaustive enumeration is necessary. This self-validation concept is inherent in the building block approach described in Section 3.5.1. If that approach is used by a certification procedure, then to a large extent it can claim that its test results are necessarily accurate.

An external approach to accuracy validation is to run the certification procedure on a golden standard MIL-STD-1750. Any procedure that runs correctly, that is without error, is by definition accurate since a golden standard is by definition error free. In practice, of course, standards with the complexity of a computer must always be viewed as possibly having errors, and thus validating a test procedure with a standard will be a trial and error process which converges to an acceptable level of confidence. In conducting a validation as well as in running

a certification procedure on a UUT, it is very important that a careful and thorough analysis of error findings is conducted. The conclusion may be that the UUT has a flaw, but it must be considered that the test may be in error or that MIL-STD-1750 may be ambiguous with respect to the test and the UUT.

### 3.4 Certification Procedure Control

Issues of certification procedure control are central to actually implementing and executing the certification test. Included are such concerns as: how is the UUT initialized, what are the operator requirements, what control software is required beyond the basic test programs, what is the procedure for loading the test programs and data, and what is the method for retrieving final results. In the sense that they are matters beyond the scope of MIL-STD-1750, these control issues are conceptually distinct from the basic design of test programs and data. But depending on the extent to which control is inherent in selfcontained programs resident in a UUT or is distributed between those programs and a test control computer, the appropriate delegation of control is an integral part of the certification procedure design. This section will propose a set of control functions which must be available manually or under program control, discuss a range of methods for initializing the certification procedure, and then outline several scenarios of control methods applicable during the certification process.

#### 3.4.1 Control Functions

The following is a list of control functions which must be available to any certification procedure.

- . Bootstrap Program Load
- . Processor Reset
- . Processor Start at Address Given



- . Processor Halt
- . Register Load
- . Register Read
- . Memory Load
- . Memory Read

These basic functions are likely to be available on every MIL-STD-1750 implementation, since they would be required in most cases for normal development of a computer. While the last four (or six) functions are available as individual instructions in the UUT, the first two (or four) are functions beyond MIL-STD-1750. Whether the functions require manual intervention, are implemented in firmware, or are supported by software is not theoretically significant, as long as they are available. Normally they would be implemented using switches on the computer maintenance panel (e.g., START, STOP, MASTER CLEAR) and special purpose firmware (e.g., a bootstrap loader in ROM). The control functions could also be made available across a standard communications channel to a separate computer by developing software protocol handlers for both computers. Standard channels which might be considered in pursuing this approach are RS-232C, MIL-STD-1553B, and MIL-STD-1397. A third option for program control is a channel with special control hardware, as is available with the AN/AYK-15A PMIU (SA 701 311) and the Users Console (SA 301 310).

#### 3.4.2 Procedure Initialization

As indicated above, MIL-STD-1750 does not specify the mechanisms for processor initialization and program loading. As key aspects of initializing the certification procedure, these capabilities require some consideration. There is some indication in MIL-STD-1750, although incompletely specified,

that the power down, power up sequence is of interest. If so, then it is important to clarify what the state of the machine is after power up. The control function of processor reset is essential to initialization because it would clear specific registers to a known state.

The options for program and data loading are those mentioned above: a bootstrap load from peripherals local to the UUT, loading across a standard channel from a second computer relying on software handlers in both, or loading across a special channel which has hardware control features. The first two options require the ability to load code into the UUT separately from any test computer. This could be done manually with panel switch entry, from a ROM, or with UUT peripherals like magnetic tape utilizing a resident loader. The third option would not require manual intervention once the processor is powered up and reset.

### 3.4.3 Control Methods

The basic control method scenario is one in which the test program is written to run by itself on the MIL-STD-1750 UUT. Although some hardware resources must be available to load the programs and data, and to indicate any errors upon test completion, control of the procedure is an integral part of the test programs within the UUT. Within this scenario, there may be a range of control complexity from a procedure which runs unattended, to a procedure which allows considerable operator interaction. For example, the AFAL Acceptance Test Program (SA 421 206) for the AN/AYK-15A contains a control executive which allows the operator to specify which test program modules to run, whether they should be run in sequence iteratively, and what to do when an error is

detected. The various test modules set flags for errors encountered, and the error flags are examined by an error processing routine at the conclusion of each test module. An interrupt handler sets flags so that each test module can monitor expected and unexpected interrupts. The Sperry Univac ATP (1979) for the AN/AYK-15A also fits this control scenario, but utilizes a different error reporting scheme: the program halts when an error is detected, providing with the halt address a pointer into the heavily documented test code.

Another scenario of control methods introduces a second computer and a standard communication channel as a means of distributing the control of the certification procedure. With this approach, the control executive of the Acceptance Test Program would be moved to the test control computer, which would load each test module and process the error flags to produce error reports. More of the control aspects of test program code are assumed by the control computer, with code in the UUT becoming more restricted to what is relevant to the function being tested. This control scenario also supports the technique mentioned as a use of a golden standard, where a test program is run in parallel in the UUT and in a golden standard. The control computer would compare the results of the two programs, marking any discrepancies as potential errors in the UUT. A third possible control structure has the control computer sending one or more instructions at a time across the channel to the UUT, providing boundary conditions first, then transferring control to the few UUT instructions, and evaluating the machine state after each test. Control is mostly with the test computer as it compares the state of the UUT with known results.

A third scenario is like the second, but elevates the communication channel to a special purpose data and control interface. Such an interface would allow the test control computer to directly access memory and registers of the UUT as well as control and monitor its performance, such that no software protocol handlers or initialization procedures would be required for the UUT. The control computer would contain all control software and would have the capability of replacing or simulating all functional components of the UUT, to isolate a particular MIL-STD-1750 function for test. For example, the test computer could replace the memory, registers, I/O, and/or load and store to memory operation of the UUT. In this way, minimal functional elements of the UUT are employed during each instruction or function test, with all other elements being substituted for by corresponding simulated functions which are known to operate correctly, then any errors detected are easily pinpointed in the UUT.

### 3.5 Certification Program Design

In discussing the matter of how complete the certification procedure could reasonably be made, Section 2 proposed a functional analysis of the 1750 architecture which could be used as the basis for organizing the test programs. There are, however, several additional program design issues. They include the necessity of ordering test programs, the desirability of having some level of complexity for the contexts in which to verify instructions, and the value of employing certain coding techniques in writing the test programs. One tradeoff that arises in considering test program design is whether the test should consciously focus on diagnosis, in terms of isolating

particular errors found, or should aim at certification only, in the sense of just detecting errors.

### 3.5.1 Organization Factors

A central issue of program organization is the matter of whether the testing of instructions and functions need to be ordered using a building-block approach, or whether it is reasonable for purposes of certification to assume that all instructions and functions not under test are correct. An ordered test organization for individual instructions would incrementally test a few basic instructions at a time, using only previously tested instructions in successive tests. Carried to an extreme, the ordering would possibly employ repeated ADD's to test a MULTIPLY, and lower level logic instructions to test the ADD. Since untested instructions are being examined one at a time, a diagnosis of the instruction causing any error is readily accomplished.

This building block approach also suggests a test ordering in terms of functional capability. For example, verification that all registers are addressable and can contain all possible values would logically precede the verification of instructions which use the registers to store their operands. After confidence is developed in a set of known capabilities, later test segments are able to utilize fairly sophisticated coding methods which are efficient, compact, and representative of instruction mixes which may be found in mission software. An ordered test organization does require a careful design of the certification procedure to take into account the dependencies of instructions for functions.

The opposite alternative is to assume that the UUT is essentially fully operable, and to focus the certification procedure on subtle errors in design or implementation. This approach assumes the correct operation of all instructions which are not specifically under test and no ordering is required. Subtle errors which exist are just as likely to be encountered as in the ordered approach, but the ability to identify where the error occurred is reduced, because the error may well be in the code surrounding the instruction being tested. The focus of this approach is certification only, at the expense of diagnosis. However, this unordered approach not only enjoys all the advantages of allowing sophisticated coding methods, it considerably simplifies the design of the entire certification procedure, since a regular format can be imposed on all instruction tests without regard to test order. And it may be concluded from real-world experience that machine errors are frequently found in unpredicted conjunction with tests designed to discover completely different problems.

### 3.5.2 Complexity Factors

Another major issue which affects the test program organization is the nature of the instruction sequence which serves to verify a given instruction. In particular, the question arises as to whether in testing an individual OPCODE, the simple context of setting up operands, performing the operation, and verifying the results constitutes an adequate test of the instruction. The concern is that architectural flaws might exist that relate to performing one operation immediately after another operation that uses common hardware. Such an error is important to find because it would affect the ability of the UUT to perform real applications software.

One basis for this concern is the notion that a complete certification procedure should offer, in addition to a thorough verification of functional performance, some confidence that a certified computer would perform correctly under mission conditions. It is known that design/implementation errors are often of the sort that are evident only under conditions of complex software operations, particularly with concurrent interrupt and I/O processing. The problem suggests two possible answers: use some real applications software as part of the test procedure and/or try to simulate an instruction mix representative of actual software through random generation of instructions or by writing sample programs which bring together archetype segments of mission software.

In practice, the use of actual mission software would be difficult since such programs usually depend on real-time data acquisition and computer response. However, it might be worthwhile to examine some operational software to aid in developing the basis for sample programs, if not to discover code segments which could be used for test purposes. In either case, what is expected to be gained from this selection is test programs which offer a diverse mix of instructions, including, for example, a mix of long and short formats or a high density of repeated arithmetic, bit manipulation, or other category of instructions in a branching control structure. Such programs can offer a more complex testing context than the more basic portion of the certification program.

The use of actual or sample mission programs satisfies a certification need without offering much diagnostic power. Because the program is designed to manipulate data, to produce a particular functional result, any machine errors which are encountered would cause program failure in an unpredictable fashion.

It is thus a kind of go/no-go test which may be useful for certification, but which tells little about the nature of any errors. It is also possible that multiple faults might cancel each other, thus masking completely an existing error condition.

Another method of creating a more complex context for instruction testing is that of executing a randomly generated sequence of instructions, limiting perhaps jumps, branches, and store instructions to a constrained address space. Such a test is not representative of any actual code sequences, but it does have advantages in simplicity of generation and the ability to run indefinitely long sequences. The test could be diagnostic in nature, stopping at a machine fault, or running for a long time as evidence of there being no errors. By using a golden standard, the test could run the random sequences in parallel, comparing the final results from the two runs. And with a mechanism for controlling the number of instructions executed in the UUT, the test procedure could take snapshot comparisons of the current UUT machine state with that of the golden machine state.

### 3.5.3 Coding Techniques

A final area of test program design that merits some discussion is the software engineering of code generation. What coding techniques will aid in producing a well-structured certification procedure which is reliable, readable, and extendable, as well as straightforward to generate from design specifications. Two techniques which seem to be particularly relevant are: one, that the program and data be physically independent, with the program referencing the data as a table or array; and two, that the program code be regular and repetitive in structure, making use of loops to access data and modify variables, and of



macros or other meta-language constructs to express repeated structures with variable portions.

The notion of separating the test code from the test data offers a number of benefits, including easy access to the data and the possibility of efficient and modular code. This technique applies primarily to the testing of individual instructions, where one or two operands are operated on to produce a known result. Structuring of data tables containing operands and results produces test programs which are inherently easy to read, update, and extend. The emphasis on readability implies an increased ability to validate by inspection the comprehensiveness of the test program.

The benefits that accrue in allowing efficient and modular code come from the necessity of referencing a table of data in a regular fashion as with a loop structure. Loops can provide a compact, efficient mechanism for accessing a large amount of data. Loops are also an effective method of reducing memory requirements when used to generate code sequences by varying data fields within particular instructions. For example, it might be desirable to have a sequence of instructions which loads and tests a particular register apply to many registers. This could be done without duplicating the sequence physically many times, by placing the sequence in a loop which, using bit manipulation instructions, modifies the actual instructions to change the in-line register references. Use of this type of self modifying or self generating code can be justified on the basis of a large savings in memory and test execution time since there are no requirements for reentrancy or fail-safe operation.

Various coding techniques are employed in part of make tradeoffs in program and data size, and thus speed of loading the test, and execution speed. There is also a tradeoff between code complexity and ease of code generation. Macros are a method of easily generating code sequences with regular changes. And, although

extensive use of macros tends to produce large programs, the regular structure of the code means high readability and reliability, since the shorter sequence in the macro definition is more likely to be written correctly and is more readily debugged.

### 3.6 Certification Test Data Design

The discussion in previous paragraphs of coding techniques which involve separating data from code was aimed primarily at testing the arithmetic and logical operations available in the MIL-STD-1750 instruction set. Such tests take as input one or more operands, perform the indicated operation in the UUT, and compare the computed result with an expected, known good result. This section identifies minimum requirements for selecting operands to use as input data and then proposes several methods which, functioning as 1750 golden standards, might be used to generate expected results from those operands.

#### 3.6.1 Minimum Requirements for Operands

Each operand to be used in a calculation can be characterized by its data format as defined in MIL-STD-1750. Data formats are defined explicitly for single precision fixed point, double precision fixed point, floating point, and extended precision floating point operands. Careful selection of data values, combined with specific knowledge of circuitry used to implement a particular function is often used by hardware manufacturers to define a limited subset of operands which will thoroughly test a particular function. In the case of MIL-STD-1750 certification, such detailed implementation details are not always available, and therefore extensive (or even exhaustive) testing may be required for specific operations. Extensive sets of operands can be randomly selected to assure a wide distribution

of operand values even when the range of possible values is large. Certain values, however, should always be included in the operand set even if the rest of the data is randomly selected.

A minimal set of operand values for arithmetic operations would include:

- . zero
- . the largest positive number
- . the smallest positive number
- . the smallest negative number
- . the largest negative number

These values represent all arithmetic boundary values, which when used in all possible combinations will guarantee the generation of overflow and underflow conditions. They also represent all four quadrants so as to generate all appropriate tests of sign manipulation.

Floating point operands should include all combinations of the above values in both the mantissa and exponent fields (i.e., a total of 25 values would be required). In addition the exponent fields should contain a set of values that result in the difference of the values being equal to the boundary conditions described above, and also close to the number of bits in the mantissa. This will force testing of scaling operations that are performed during floating point add/subtract type operations.

For logical or bit manipulation operations, a minimal set of operand values would include:

- . all bits zero
- . all bits one
- . bits alternating one and zero

. bits alternating zero and one

These values represent the logical operation boundary conditions with all zeros and all ones, and the independence of bits with the alternating patterns. Used in all combinations of pairs, they will set all relevant states of the status word and confirm all bit changes, i.e., 1→0, 0→1, 1→1, 0→0.

### 3.6.2 Generation of Expected Results

Each functional test or sub-test requires a unique set of expected results for the operand or operand pairs to be tested. The number of results to be tested can be extensive, even if only a few operands are tested. For example, if only ten operands are in each of two operand lists, then 100 results must be checked if all combinations are used for each calculation that uses two operands. The total number of results that must be checked will likely be uneconomical with manual calculation. Therefore, automated techniques for generating correct results must be available. And since these techniques are performing 1750 operations, they must represent 1750 golden standard devices.

The most obvious method of generating expected results is to use a complete 1750 implementation in which to run test code that performs the indicated operation on the input set of operands. This approach could use existing 1750 hardware--at least two machines are now available--or it could use a 1750 software simulator--one is scheduled to become available to SEAFAC soon. The method has other advantages, including the usefulness of actual test code and the ability to provide all necessary results of the operation: the results, the condition code, and the occurrence of any interrupts.

An alternate approach to the same end is to write a stand alone 1750 arithmetic and logical golden standard simulator on a separate computer with accuracy that equals or exceeds the requirements of MIL-STD-1750. This simulator could simply calculate results in the format of the host computer and convert the numbers to 1750 representation, with appropriate tests for overflow, underflow, and condition code. This method is attractive in that a large amount of data can be generated with a minimum of programming effort.

A third approach is to rely on a 1750 golden standard capability embedded in an assembler used to generate the test programs. This technique is conceptually simple, as can be seen from the example in Figure 3-2. It has the disadvantage of not providing for all the required results, since no condition code or interrupt information is available at assembly time. However, if such an assembly capability were available, it would provide an alternate method of calculation which would be useful as a validation procedure.

### 3.6.3 Storage of Operands and Expected Results

The expected results for an instruction or series of instructions under test must somehow be incorporated into the test procedures themselves. The method of incorporating the expected results into the test procedures will depend heavily upon such factors as: when the expected results are generated relative to the time they are used for verification or results, and how the expected results are compared to the results provided by the UUT.

Tables of expected results can be generated in core image format, transferred to a storage medium, and loaded into the UUT for comparison against actual results by programs residing in

FIGURE 3.2. USE OF ASSEMBLY TIME OPERATIONS  
TO GENERATE EXPECTED RESULTS

OP1	EQU	1	
OP2	EQU	-32768	
	:		
	:		
DATA2	+	OP2	
	:		
	:		
	LIM	R1,OP1	. load first operand
	A	R1,DATA2	. ADD to second operand
	CIM	R1,OP1+OP2	. compare to expected results
	JC	NE,ERROR	. exit if error detected
	:		

the UUT. Alternatively, expected results can be converted to symbolic data in a format that is acceptable to the MIL-STD-1750 assembler and incorporated directly into the test program residing in the UUT. Yet another scenario would transfer results computed in the UUT to the test computer for comparison.

### 3.7 REPORT GENERATION PROCEDURES

A full and complete report of the results obtained during an attempt to certify a particular implementation of MIL-STD-1750 should be provided. This report is required to document the fact that an attempt to certify the implementation was made, and to identify specific test failures that may have occurred. In addition, any results that may differ from implementation to implementation due to inclusion or exclusion of specific options, or other ambiguities allowed by the specifications should be reported. The following paragraphs outline procedures, data formats, and information content of various reporting possibilities.

#### 3.7.1 Methods of Generating Reports

Report generation procedures may vary from simple manual methods to complex computer generated description and analysis reports. Choice of a report generation method depends strongly on the level of detail required. A simple pass/fail report can easily be generated by manual techniques; a report that includes extensive data that can be used for failure analysis should be automated so that extraneous errors and/or loss of diagnostic information will not be introduced by clerical errors.

Compromise methods of generating reports such as handling all pass conditions manually and utilizing hand edited memory dumps

for reporting test failures should be given careful consideration because of the low costs of such a compromise. More complex failure analysis performed by a computer would probably allow (require) more sophisticated error reporting.

### 3.7.2 Content and Format of Certification Reports

Any certification report, regardless of how it is produced, should include certain minimum information about the certification process.

This information includes:

- . The manufacturer, model, and serial number of the unit under test.
- . The date and time the test was started/completed.
- . Identification of any previous test reports that were issued for the specific model under test.
- . The revision of MIL-STD-1750 to which the certification is being performed.
- . The revision level(s) of the certification test programs and procedures.
- . The project, system, or subsystem for which the unit under test is being certified (i.e., the end user).
- . The names and affiliation of individuals witnessing the certification process.



- . The results of the test at least to the level of pass/fail.

Additional information that would be helpful includes a list of all possible tests (or subtests) that are included in the certification process annotated as shown in Figure 3-3. The advantages of this type of reporting are:

- . Specific identification of all tests that have been executed is provided.
- . Each test that failed is identified.
- . The MIL-STD-1750 instruction or function relevant to the test is identified.
- . Results of action taken to determine the nature of any failure are identified.

### 3.7.3 Detailed Failure Analysis

Many automated failure analysis reports can be generated if sufficient error information is retained by the certification test programs. The information requirements will vary with the structure and content of the particular test component that detects the error as well as the level of analysis desired. Figure 3-4 shows a sample report that indicates one type of reporting possible. This report could conceivably be used to detect failures of the following architectural entities:

- . OP CODE (Functions)
- . Registers
- . Addressing Modes

PROCESSOR AND (EXECUTIVE CONTROLLED)										UNIVAC 15A STATUS SHEETS									
DATE: 11-SEP-62, TIME: 07197, PAGE: 03 OF 221																			
TEST	YES/NO	IF	IF FAILED	IF	IF FAILED	ISA	ATP	SPEC	REMARKS										
ID	NAME	DATE	PASS/FAIL	PROBLEM	PROBLEM	PROBLEM	PROBLEM	PROBLEM	PROBLEM										
01A	UN	102-12-00	PASSED																
01B	UN	102-12-00	PASSED																
01C	UN	102-12-00	FAILED							Errors result from dividing a large number by a very small number. The expected results are different.									
01D	UN	102-12-00	FAILED							Errors result from dividing a large number by a very small number. The expected results are different.									
01E	UN	102-12-00	FAILED							Errors result from dividing a large number by a very small number. The expected results are different.									
01F	UN	102-12-00	FAILED							Errors result from dividing a large number by a very small number. The expected results are different.									
02M	FA	102-12-00	PASSED																
021	FA	102-12-00	PASSED																
022	FA	102-12-00	PASSED																
023	FA	102-12-00	PASSED																
024	FA	102-12-00	FAILED							Possible ISA problem. Results under investigation.									
025	FA	102-12-00	FAILED							Possible ISA problem. Results under investigation.									
026	FA	102-12-00	FAILED							Possible ISA problem. Results under investigation.									

Figure 3-3. Sample Page from AFAL AN/AYK-15A ATP Report

- . Operand Types
- . Memory Addressing
- . Memory Data

Automated reporting of this type can have the disadvantage of producing mountains of data that requires further analysis to determine the specific architectural component(s) that failed. Summary reports can perform this valuable task as shown in Figure 3-5. In this example, the information presented in the detailed report is summarized in terms of various distributions of the failures. Careful examination of these distributions can provide an indication of specific malfunctions even when the errors have a multiplicity of symptoms.

DOCUMENT NO.  
PX 13243

SPERRY  UNIVAC

FIGURE 3-4. INSTRUCTION CODE TEST -- DETAILED FAILURE REPORT

OPCODE  
RA register  
RX register  
Label field  
Operand type  
(RA)  
(RX)  
DA  
DO  
Results  
Expected results  
Failed bits  
Expected status  
Received status  
Expected interrupt  
Received interrupt

DOCUMENT NO.  
PX 13243

SPERRY UNIVAC

FIGURE 3-5. INSTRUCTION CODE TEST --- SUMMARY FAILURE REPORT

Frequency of Failures for \_\_\_\_\_ Test Failures

<u>OPCODE</u>	<u>% Failures</u>	<u># Failures</u>
-	-	-
-	-	-
-	-	-
<u>RA Registers</u>	<u>% Failures</u>	<u># Failures</u>
<u>RX Registers</u>	<u>% Failures</u>	<u># Failures</u>
<u>Bits in RA</u>	<u>% Failures</u>	<u># Failures</u>
<u>Bits in RX</u>	<u>% Failures</u>	<u># Failures</u>
<u>Bits in DA</u>	<u>% Failures</u>	<u># Failures</u>
<u>Bits in DO</u>	<u>% Failures</u>	<u># Failures</u>
<u>Error bits in Result</u>	<u>% Failures</u>	<u># Failures</u>
<u>Error bits in Status</u>	<u>% Failures</u>	<u># Failures</u>

#### 4. CERTIFICATION PROCEDURE EVALUATION

The purpose of this section of the report is to raise and discuss the key issues involved in evaluating various test methodologies. Section 4.1 defines what a test methodology is in terms of its attributes, and discusses the methods of combining alternatives to each of these attributes. Criteria for evaluating test methodologies are discussed in section 4.2 and a system efficiency model is developed in section 4.3. Subsequent sections expand upon previously introduced evaluation criteria until certification procedure design goals are introduced in section 4.7. Finally, an evaluation of currently known certification procedure options available to SEAFAC are introduced in section 4.8. The overall effect of the discussion of test methodologies and evaluation procedures is to define relations between factors such as risk, cost, reliability, and completeness. Specific models are proposed which are used generatively to develop design goals that are achievable and cost effective.

##### 4.1 Test Methodologies

A test methodology can be described as a collection of procedures, and data used to 1) provide test programs and input data for those programs, 2) execute those programs on any 1750 computer (subject to the constraints defined in section 2.4.2) and 3) evaluate the resulting output data to determine whether the UUT complies with MIL-STD-1750. In order to evaluate and compare proposed test methodologies, each one must be thoroughly and unambiguously described in terms of its attributes. The following attributes are considered germane to the specification of a test methodology.

1. The design of test programs
2. The design of test data

3. The method of evaluating test programs and data for correctness
4. The method of loading and initiating test programs and associated data structures
5. The method of recovering, analyzing and reporting test results.
6. The method of adapting the test methodology to optional features and/or changes in MIL-STD-1750

A great number of possible test methodologies can be derived by listing alternative procedures and data for each of the attributes mentioned above. If only two alternatives were available for each of those six attributes, then a total of sixty-four (64) possible test methodologies would exist. Fortunately, most of the alternatives selected for one attribute will limit the possible selection of other attributes. Therefore, the total number of reasonable methodologies is much smaller than the number that can be derived by simply combining attributes.

Evaluation of alternatives within each attribute category cannot be used to select a whole test methodology because they cannot be combined arbitrarily. But, separate evaluations can be used to measure the effects of applying more than one test methodology to the certification problem. For example, testing of the ADD operation might not be considered complete if only boundary values are used for testing; also it might not be considered complete if only a few random values were used. Application of both techniques however might provide an adequate level of completeness.

#### 4.2 Evaluation Criteria

Each test methodology and, indeed, each attribute alternative can be evaluated with respect to various criteria. The three

major evaluation criteria are discussed in this section; test efficiency, program reliability and cost impact.

#### 4.2.1 Efficiency

Each attribute, alternative defines parameters that can be used to determine overall test efficiency with respect to total test time and/or memory requirements. Efficiency measures can be estimated by counting UUT memory references used to execute a particular test sequence. The functional efficiency of the test code is defined as the ratio of the number of memory references required for the instruction(s) being tested to the total number of memory references used for setup, execution, and examination of results.

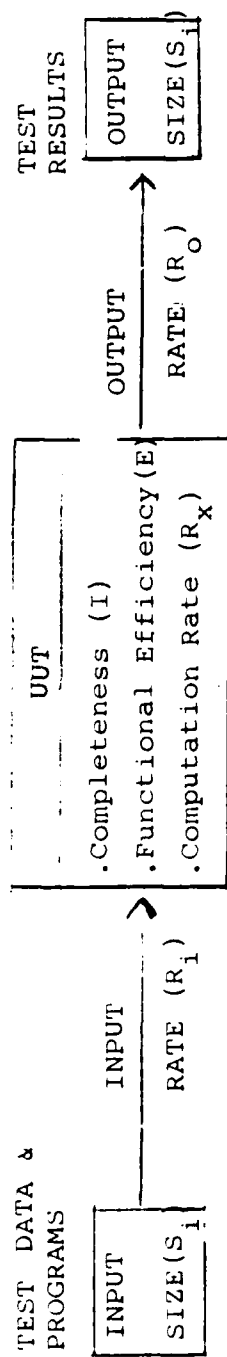
This functional efficiency and several physical constants of the unit under test define the time vs memory trade-off and can be used to compute the total time required to execute a certification test component for a given unit under test by proper application of the system efficiency model shown in Figure 4.1. This model shows that the total test time is the sum of the time required to load the test and test data,  $S_i/R_i$ , the time to execute the code,  $(I/(E \cdot R_x))$ , and the time taken to retrieve the results,  $S_o/R_o$ .

#### 4.2.2 Reliability

Reliability of the certification process can be measured largely in terms of the completeness, clarity, and verifiability of the test code. If a particular test methodology tests more unique machine states than another, then it is more complete. If a particular segment of test code is more straightforward or more easily understood than another, then it has greater clarity. If one certification procedure utilizes more



Figure 4.1. System Efficiency Model



-----Test Program Parameters-----		
S <sub>1</sub> - input size	I - No. instruction words tested (completeness)	S <sub>0</sub> - output size
	E - Functional Efficiency	
-----Shared Parameters-----		
R <sub>1</sub> - input rate		R <sub>0</sub> - output rate
-----1750 PARAMETERS-----		
R <sub>x</sub> - Execution Rate		
-----Time Equation-----		

$$T = \frac{S_1}{R_1} + \frac{I}{E \cdot R_x} + \frac{S_0}{R_0}$$

independently produced data to validate the process then it has greater verifiability.

#### 4.2.3 Cost

Each attribute alternative will have associated requirements for hardware facilities, support software facilities, skill levels, amount of operator involvement, and other costs associated with development or use of the certification component. These requirements can be rank ordered for each attribute such that the highest cost alternative is given a rank of one, the next highest cost a rank of two and so on. Alternatives that have nearly equal costs (within 25%) will be given equal ranks.

#### 4.3 Application of the System Efficiency Model

The system efficiency model shown in Figure 4.1 can be used to conceptually define the most time efficient test procedure. To do this the time equation:

$$T = \frac{S_i}{R_i} + \frac{I}{E \cdot R_x} + \frac{S_o}{R_o}$$

must be minimized with respect to one or more of the input parameters.

##### 4.3.1 Simplifying Observations

Several practical observations will be helpful at this point. First, the data transfer rates into and out of the UUT will very likely be of similar speed. For example, if a magnetic tape unit is utilized for program and data loading then it can also be used for logging of the test results. If a test computer link is used to transfer data to the UUT then it will

probably be used for collection of results. This observation allows the time equation to be simplified to two terms:

$$T = \frac{S_i + S_o}{R_L} + \frac{I}{E \cdot R_x}$$

where  $R_L = R_i = R_o$

A second simplifying observation is that two alternatives exist for the location of known good results. Either the known good data resides within the UUT or outside the UUT. In either case, the amount of data transferred into or out of the UUT will be the same, assuming that results are similarly coded in either case. This simplifying assumption allows us to rewrite the time equation again.

$$T = \frac{S_L}{R_L} + \frac{I}{E \cdot R_x}$$

where  $S_L = S_i + S_o$

A third simplifying observation is that for a wide variety of test programs,  $S_L$  is directly proportional to the number of instructions being tested. In order to better demonstrate this relationship, two examples are presented below.

Case I. Non-looping ADD Test with Comparison in UUT

		NUMBER OF MEMORY REFERENCES
LIM	R1, IOPl	2
A	R1, MOP2	3
CIM	R1, RESULT12	2
JC	NE, ERROR	2

NUMBER OF MEMORY  
REFERENCES

LIM	RI, IOP5	2
A	RI, MOP7	3
CIM	RI, RESULT57	2
JC	NE, ERROR	2

In this example each test of the ADD instruction requires eight memory references for program code plus one additional reference for each operand of the add instruction, bringing the total memory requirement to nine cells. The total memory requirement is therefore nine cells per tested instruction. The only requirement for output is a single error flag. The total execution time for testing 100 add instructions with case I is approximately:

$$T = \frac{100 \times 9}{R_L} + \frac{100}{E R_X}$$

The definition of the functional efficiency E as the number of memory references in the tested instruction divided by the total number of memory references allows us to estimate the total execution time in terms of the data link transfer rate,  $R_L$ , and the UUT execution rate,  $R_X$ .

$$T = \frac{100 \times 9}{R_L} + \frac{100}{(3/9) \cdot R_X}$$

Case II. Looping ADD Test with Comparison in Test Computer

NUMBER OF MEMORY  
REFERENCES

	LIM	R4,M*N	2
	LIM	R2,M	2
OUTER	LIM	R3,N	2
INNER	L	R1,OPERAND,R2	3
	A	R1,OPERAND,R3	3
	S	R1,RESULT,R4	3
	AISP	R4,1	1
	SOJ	R3,INNER	2
	SOJ	R2,OUTER	2

In this example, each test of the ADD instruction requires slightly over twelve memory references, yielding an efficiency of  $E = 3/12$ . The number of memory cells transferred for 100 test instructions is:

20 cells in program
+10 cells in OPERAND Table
+100 cells for storage of results
130 cells total

The total execution time for Case II is:

$$T = \frac{130}{R_L} + \frac{100}{(3/12) \cdot R_x}$$

The number of words transferred in Case I and Case II is bounded by a linear function of the number of individual instructions tested,  $I$ , and consists of:

- . program code that may be either a linear function of  $I$  (Case I) or a constant (Case II).
- . operands that may be included in the code (Case I) or stored in tables (Case II). Use of a single table for both operands of a two operand instruction can reduce the number of operands required to be proportional to

the square root of  $I$  as in Case II.

- expected results are a linear function of  $I$  since each execution of an instruction produces a consistent and finite number of results. (Specific exceptions to this rule do exist. e.g., the MOV instruction or algorithmic self test methods (Section 3.2.3)).

The fourth and final simplifying observation deals with typical values for  $R_L$  and  $R_x$ . Current memory technology and implementation techniques dictate a range of values for  $R_x$  that are centered around  $R_x = 10^6$  references per second with a range of about  $R_x = 10^5$  for slow microprogrammed processors to  $R_x = 10^7$  for pipeline processors with relatively fast semiconductor memory.

The value range for various data links is as follows:

<u>Data Link</u>	<u><math>R_L</math> (words/second)</u>
DMA channel	$1.0 \times 10^6$
MIL-STD-1553B	$5.0 \times 10^5$
Magnetic Tape	$1.0 \times 10^4$
RS-232 (9600 baud)	$4.0 \times 10^2$
Slow paper tape	$1.0 \times 10^1$
Manual entry	$1.0 \times 10^{-1}$

With the exception of directly coupled memory links,  $R_L$  is very much smaller than  $R_x$  and can be considered the major efficiency factor until the product of the functional efficiency,  $E$ , and the execution rate  $R_x$  becomes small with respect to  $R_L$ . To illustrate, assume values of  $R_L = 10^4$  and  $R_x = 10^6$  in the time equations for Case I and II:

Case I

$$\begin{aligned} T &= \frac{100 \times 9}{1 \times 10^4} + \frac{100}{(3/9) \times 10^6} \\ &= 9 \times 10^{-2} + 3 \times 10^{-4} \\ &= 9.03 \times 10^{-2} \end{aligned}$$

Case II

$$\begin{aligned} T &= \frac{130}{1 \times 10^4} + \frac{100}{(3/12) \times 10^6} \\ &= 1.3 \times 10^{-2} + 4 \times 10^{-4} \\ &= 1.34 \times 10^{-2} \end{aligned}$$

In each of these equations, the error introduced by completely dropping the process execution time is negligible.

The third simplifying assumption above was that the amount of data transferred,  $S_L$ , and the number of instructions tested,  $I$ , were functionally related. There is, however, a class of test programs for which this relation is not apparent. Tests in this class might be considered as functional component tests where the efficiency model described applies at the level of a particular functional component within the UUT, but does not apply to the larger system consisting of the UUT and its environment. An example of this type of test program is a memory test.

A memory test confirms that a set of test data can be loaded into registers, transferred to the memory unit under test, retrieved, and tested against the register contents. In a more formal notation, a possible memory test is:

$$\begin{array}{llll} & & b & b \\ m_i & f_k(i) & i = a & \\ & & b & \\ M_i : f_k(i) & & i = a & k = m \end{array}$$

These equations show that a single program containing  $n-m+1$  different data functions and testing  $b-a+1$  different addresses can test  $(n-m+1)$  times  $(b-a+1)$  different machine states. In this case changing the address limits,  $a$  and  $b$ , does not affect the size of the program that must be transferred to the UUT, but it does have a significant effect on the execution time of the program.

The point is that for this class of test programs, the total number of instructions transferred to the UUT is very small with respect to the number of states tested and is therefore not limited by the data rate,  $R_L$ . Other program relationships, however, remain in effect. Specifically, total test time is a function of completeness and the functional efficiency of the test code.

#### 4.3.2 Implications of the Model as Design Criteria

The following major conclusions can be drawn from the system efficiency model. These conclusions can and should be applied as design criteria for generating certification test procedures and/or evaluating the efficiency of test designs.

1. The total execution time required to perform tests of the type described is controlled by the mechanism for transferring programs and data, therefore use of manual techniques for entering or retrieving information should be avoided.
2. The amount of data transferred (and therefore the execution time) for programs that test the correctness of results in the UUT is the same as that required for determination of correctness outside the UUT. Therefore, by performing checks within the UUT, rather than in a locally connected test



computer, the test can be performed with a stand-alone UUT, appropriate maintenance console and load device.

3. Program loops can be used to reduce memory requirements with little impact on total test time provided that  $E/(E \cdot R_X)$  remains small with respect to  $S_L/R_L$ .
4. The monotonically decreasing risk as a function of completeness described in Section 2.1 indicates that minimizing risk involves maximizing instruction completeness by increasing the number of instructions tested (I in the system efficiency model). To do this requires as large a code and tests data base as can be effectively generated and maintained. Note that memory (long term storage) requirements are linear functions of completeness as shown in both Case I and Case II.
5. Reduction of risk as described above corresponds to increasing any and perhaps all of the following certification test parameters:
  - . the total test time
  - . the total number of instructions (or data variants) tested
  - . the functional efficiency of the test code

#### 4.4 The Role of Reliability

Test certification reliability was defined in section 4.2.3 in terms of completeness, clarity, and verifiability of test code. Section 4.3.2 related the completeness issue to the number of unique instructions tested. An extension of this idea of completeness can be related to the number of unique machine status and state transitions tested. It is obvious by simple inspection that the number of machine state transitions possible is well beyond the realm of testability, but simple architectural assumptions can be used to reduce the number of degrees of freedom of the test.

##### 4.4.1 Expanded Definition of Completeness

The major assumption to be made is that each functional entity can be individually tested. This means, for instance, that each bit in each memory cell and register can be individually tested for its ability to be set and cleared. It would also mean that each byte could be tested for its ability to contain all 256 possible bit patterns. The assumption of architecturally divisible components would allow simple reduction of the number of tests required in the following manner.

Assume that a 1024 ( $2^{10}$ ) word by 16 bit segment of memory is to be tested, and that each word is defined as consisting of two bytes. A functionally sound test might attempt to verify that all possible values can be stored in the memory. To do this the memory can be tested as a word addressable unit with each word having a possibility of assuming  $2^{16}$  different states. Under these conditions  $2^{26}$  different combinations must be tested. Note however, that by assuming functional independence of each byte the  $2^{10}$  word memory can be considered as a  $2^{11}$  byte memory with each byte having  $2^8$

possible states. To test this same memory now requires only  $2^{19}$  test combinations, less than one percent of the number required by assuming a word organization.

Note that when implementation details are unknown, as is the case in 1750 certification, then the assumption that the physical hardware is organized in 16 bit words is completely unfounded. There is no reason why the memory can't be organized as 56 bit words, with 16 bits of polynomial error checking, for example. The real reason for choosing functional entities to simplify testing is that each of these abstractions is completely defined by MIL-STD-1750.

Figure 4.2 outlines possible functional entities that can be used to reduce the number of individual tests and still guarantee a high level of completeness. This is done by varying a particular entity over its entire range while at the same time other functional entities are varied over a small sampling of their entire range. Completeness of any methodology can be measured in terms of the number of samples of each functional entity that are tested in this manner relative to the performance "standard" presented in the figure.

#### 4.4.2 Expanded Definition of Clarity

The same figure used to define completeness also simplifies and defines the meaning of conceptual clarity by identifying which test module provides the major test of a particular functional component. This level of clarity provides insight into where a particular functional entity should be tested, but it must be pointed out that it does not necessarily indicate all modules where it is used or referenced. This is important for functional entities that may undergo change or re-definition, such as the proposed change in base register assignment from general registers R4-R7 to R12-R15.

Figure 4-2.

Entities that Require Thorough Testing

## TEST

ENTITY	Non-Control Instruction	Register Addresses	Memory Addresses	Derived Addressing	Register Data	Memory Data	Jump/Br Instructions	I/O Instruction	Inter- rupts
Non-Control Instructions	275								
Register Addresses	5	16			16				
Memory Addresses	5		64K	64K		64K	64K		
Derived Addressing				64K					
Register Data	5	64K		64K	64K			64K	
Memory Data	5					64K			
Jump/Br Instructions							17		
I/O								23	
Instructions Interrupts	3								16
Number of States for Complete Test	N 2 <sup>20</sup>	2 <sup>20</sup>	2 <sup>16</sup>	2 <sup>48</sup>	2 <sup>20</sup>	2 <sup>32</sup>	2 <sup>20</sup>	2 <sup>20</sup>	2 <sup>4</sup>

#### 4.4.3 Expanded Definition of Verifiability

Section 3.3 indicated the various methods of verifying the correctness of the certification procedures. The most easily applied and easily understood methods are the majority vote and self test methods. The majority vote method is rapidly becoming an available possibility because "votes" have been cast by Sperry Univac, Westinghouse, and DAIS in the form of AN/AYK-15A hardware and test software, and the time is rapidly approaching when the Advanced Digital Avionics Module (ADAM) project will be able to provide additional votes in the form of both hardware and software.

The fact that each of the participants have independently pursued implementation and/or test of MIL-STD-1750 hardware or software, while at the same time communicating their findings to the MIL-STD-1750 Users Group to identify and resolve any conflicting views lends a great deal of substance to each vote.

#### 4.5 Certification Procedure Costs

Section 2.1 outlined the relationship of risk to completeness and Section 4.3.1 established simplified linear relationships between execution time and memory requirements with respect to test completeness. The development and/or operating costs associated with any particular test program design will normally exhibit this linear increase in proportion to the level of completeness thus allowing cost to be substituted for completeness as shown in Figure 4.3.

We know from the shape of the risk curve that there are diminishing returns in both dimensions. The level of completeness outlined in section 4.4.1 provides a reasonable upper

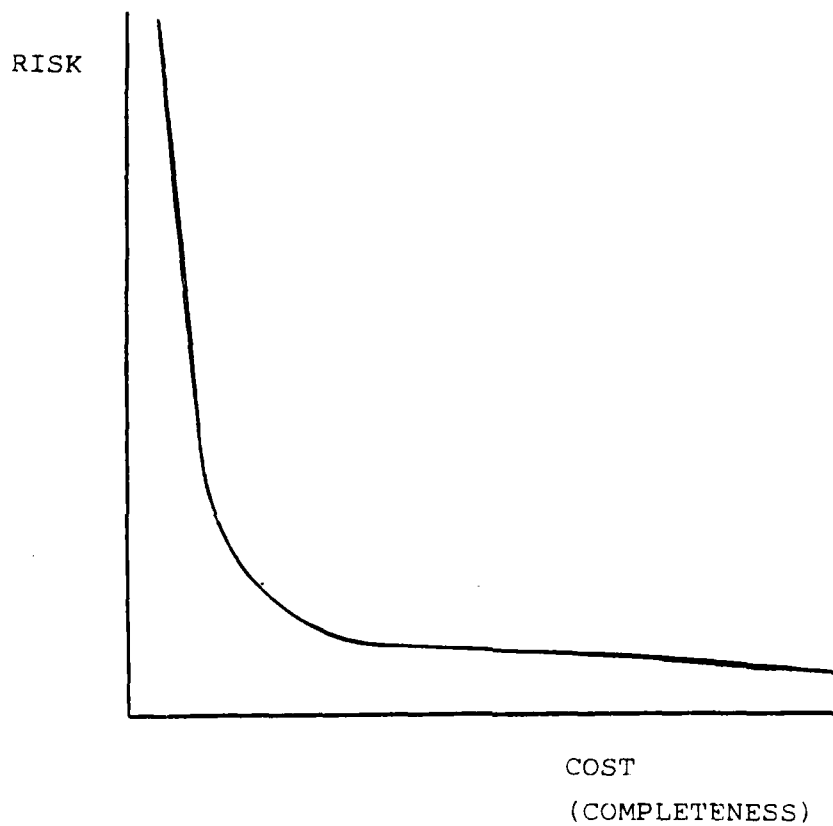


Figure 4-3. Relationship Between Risk and Cost

bound for the amount of functional testing to be done. This method of limiting test completeness establishes a linearly increasing amount of testing required to validate operation of each additional functional entity. That is, functional entities are assumed to be adequately tested if each feature is tested in the context of only a few samples of state for all other related functional entities. Thus, for example, the instruction tests cover all opcodes, but use only a limited number of memory and register addresses and data values; the address and data functions are the focus for separate tests. Any assumption other than independent testing of functions leads to either an inadequate test or an exponential increase in the number of required test states, with an associated rise in cost.

Several important cost factors should be identified at this point,

1. Using existing code is probably the least costly alternative for a given level of completeness.
2. Extending the completeness of a particular test adds to the cost.
3. Verifying existing code adds to the cost and lowers risk, but does not extend completeness.
4. Costs associated with increasing program completeness depends upon the program design. Some program designs are more easily extended than others.
5. Excessive costs may be incurred if an attempt is made to extend program completeness beyond its design limits. (e.g., a new test module may be required to completely test a feature that was partially tested in the past).
6. The most cost effective approach can only be determined if a level of completeness is specified.

#### 4.6 Design Goals

Several candidate design goals have been previously indicated, these goals will now be made explicit with references made to appropriate sections of the report. Specific recommendations for implementation of procedures that meet these goals are also given, as references to Section 5.

<u>Design Goal</u>	<u>Reference(s)</u>	<u>Recommendation(s)</u>
1. Each functional entity should be completely tested (within reasonableness constraints)	2.4 - 2.4.2,	5.6.3-5.6.7
2. Tests of context including interrupts, I/O, and processing should be included	3.5.2	5.6.8, 5.6.9
3. The system should be easy to use; complex operating procedures should be automated	3.4.3, 4.3.1	5.2 - 5.4, 5.11, 5.13
4. The amount of test data should be easily expanded, preferably by an automated process	3.6.2, 3.6.3	5.7 - 5.7.3
5. The certification process should itself be validated	3.3 - 3.3.2	5.8 - 5.8.3
6. The certification process should be adaptable to specific options and/or changes in MIL-STD-1750		5.9 - 5.10
7. The certification facility should provide a means of generating, testing, and archiving test procedures	3.1 - 3.1.2	5.2, 5.5, 5.12



#### 4.7 Evaluation of Candidate Methodologies

Existing or planned test procedures can now be evaluated relative to the design goals just specified. Normalized rank ordering of alternate methodologies against each design goal has been chosen as a measurement technique. This technique allows subtle differences in quality to be differentiated on the basis of rank, and limits the dominance of any wide disparity found for one comparison of candidate methodology and design goal.

##### 4.7.1 Description of Candidate Methodologies

Two broad options are available for selection of test methodologies. First, an existing test such as the AFAL (DAIS) ATP, described in Appendix F, and/or the Sperry Univac ATP, described in Appendix G, can be used as a baseline for producing the desired test. The second option is to develop new test programs and procedures that meet the design goals enumerated in Section 4.6.

A range of test procedure implementation possibilities exists for each of the two broad options. The first option would provide three major implementation possibilities:

- . Use the existing test(s) with minimum modification or extension.
- . Use the existing test(s) as a framework for a test with increased completeness
- . Use either of the above two possibilities with the addition of an independent validation of the test(s).

The second option, developing a new test, would allow reasonable design trade-offs to be made for each of the six attributes introduced in Section 4.1. A major consideration for any new

test is careful selection of the factors that affect the time vs completeness trade-off as described by the system efficiency model (cf. Section 4.3). Specifically, the level of test automation must be given careful consideration when the amount of test data and/or the number of core load modules becomes large.

The many test procedure implementation possibilities that utilize either existing baseline tests or new test programs can be represented by candidate test methodologies. Four test methodologies are considered representative candidates for use in certifying compliance with MIL-STD-1750: the existing ATP(s); the existing ATP(s) with further validation of design performance; a new test procedure that executes on the UUT without benefit of connection to a test control computer; and a new test procedure that utilizes a "standard" communication link to a test control computer.

A detailed description of the last two approaches will be found in Section 5. These methodologies represent selection of specific trade-offs based on the assumptions described in Section 2, the techniques described in Section 3, and the system efficiency model and design criteria developed in Section 4. This detailed design approach was taken so that specific functional test elements could be defined to the point where coding techniques, flow charts, and data designs could be used to reduce the number of subjective judgments that must be made when methodologies are evaluated based on more general descriptions. The following paragraphs summarize each candidate methodology.

Methodology I - Use the existing AFAL ATP and/or the existing Sperry Univac ATP. The major advantages of this option are the low cost of implementation and the simplicity of operation.

The AFAL ATP seems to be more complete and less hardware dependent than the Sperry Univac ATP, but the latter employs some desirable coding techniques, such as the use of tables for operand storage. Thus, some combination of the two ATP's might be appropriate.

Methodology II - Validate and use either or both ATP's. This method requires the development and use of a golden standard (simulator) as described in Sections 3.2 and 3.3. The advantage of this method over method I is that additional emphasis is placed on verifying that the test actually performs as intended.

Methodologies III and IV - Design a new set of test programs to meet the design goals. One possible set of programs that can meet the design goals is described in Sections 5.6 through 5.8. Those programs use a variety of techniques to achieve a high degree of completeness as discussed in Section 4.4.1. Validity of the process is assured by use of a simulator as in method II, but with the additional confirmation of correct functioning of all components cross checked by use of the AFAL ATP as described in Section 5.8.3.

Two methodologies deriving from this design are possible. The first, method III, would be manually controlled. That is, it would utilize bootstrap and/or control (or maintenance) console functions for entering programs and test data and for reporting results as described in Section 3.4.

The second methodology, method IV, would be controlled from the VAX-11/780 computer via data links as described in Sections 5.2 through 5.4.2. The advantage of this variant is that the large test data base can be automatically transferred and controlled by the VAX-11/780.

#### 4.7.2 Detailed Evaluation

A detailed evaluation of the four candidate methodologies has been performed using rank ordered scoring techniques. Use of rank ordering allows subtle differences between methodologies to be represented while at the same time preventing any one characteristic from dominating the score. Rank order scores were assigned to each of the test methodologies for each of the seven design goals and cost (cf. Section 4.6). Highest ranks are most desirable. A weighted score for each of the candidate methodologies was computed by the following process:

1. Normalize the rank for each design goal by the highest rank for that design goal
2. Multiply the normalized ranks for each design goal by the relative weight for that design goal
3. Sum the weighted normalized ranks for each methodology
4. Divide the sum calculated in step 3 by the sum of the relative weights

The scores shown in Figure 4-5 are for equal weighting of all seven design goals and for cost. A different set of test scores and perhaps even a different outcome will result if different relative weights are applied as shown in Figure 4-6. Note, however, that Method IV is exceptionally broad in scope in that it has the highest rank in seven of the eight evaluation criteria.

This section of the report summarizes the reasons for placing each of the four candidate methodologies in its respective rank as shown in Figure 4-5.

#### 4.7.2.1 Evaluation of Test Completeness

Methods I and II are given lowest and equal ranks because they both have a small amount of test data (about ten operand pairs per instruction). Method III is given the next rank because

it is capable of supporting test completeness at or near the levels shown in Figure 4-2 subject to test time as constraining factor. Method IV is given a higher rank than Method III for two reasons.

First, automation of the data transfer will eliminate many of the delays associated with manual bootstrap methods by allowing more test data to be used per unit of test time. Second, several tests, including the Random Instruction Sequence Tests and the Context Tests described in Sections 5.6.9 and 5.6.10 require the use of an external test facility capable of determining the final machine state of the UUT.

#### 4.7.2.2 Evaluation of Context Testing (cf. Section 3.5.2)

Methods I and II are given lowest and equal ranks for the following reasons. Method I and method II are equal because they represent the same initial test code. Method III includes Methods I and/or II as a test of context (cf. Section 5.6.10) and uses sequence tests in addition (cf. Section 5.6.9). Therefore, test III is more complete in context than Methods I or II. Method IV has a more complete test of context than Method III because it contains all of the tests in Method III plus an additional random instruction sequence test described in Section 5.6.9.

#### 4.7.2.3 Evaluation of Ease of Use

Evaluation of the Ease of Use supported by each of the test methodologies requires some independent judgement of quality for each methodology. The Sperry Univac ATP is designed as a stand alone test that utilizes the minimum amount of support hardware. Its use in a stand alone mode is simple and efficient. The AFAL ATP executive includes a control capability that supports interaction with a test administrator. This

manual intervention in the form of loading and/or preparing many test modules (in the range of 20-30 core loads) and of recording test results. For this reason method III is ranked below methods I and II. Method IV is the only method that specifically addresses the question of connecting a test computer to an arbitrary implementation of MIL-STD-1750. Therefore, it ranks the highest of the methods in level of automation, even though the use by the AFAL ATP of the AN/AYK-15A Users Console provides an equivalent control capability.

The AN/AYK-15A Users Console (SA 301 310) provides a hardware mechanization of the protocol and data link handler function described in Sections 5.3 through 5.4.1. This device provides equivalent and in some sense superior levels of hardware control for the purpose of certification testing; but, it does so at the expense of generality. The Users Console is an excellent, but implementation specific version of the capabilities defined for method IV in Sections 5.2 through 5.4.2. Method IV is ranked above the AN/AYK-15A mechanization of methods I and II because of its general applicability to MIL-STD-1750 testing.

#### 4.7.2.4 Evaluation of Test Expandability

Methods I and II are given lowest and equal ranks because they require examination of code segments and insertion of data into the appropriate places. Methods III and IV, on the other hand, have tables of data for easy expansion (cf. Section 5.7). Method IV is given a higher rank than method III because it is capable of generating the random test sequences which require a simulator (cf. Section 5.6.9).

DESIGN GOAL (Section 4.6)	RELATIVE WEIGHT	METHOD (Section 4.7.1)			
			II	III	IV
1. Completeness	1	1	1	2	3
2. Context	1	1	1	2	3
3. Ease of Use	1	2	2	1	3
4. Expandability	1	1	1	2	3
5. Validation	1	1	2	3	3
6. Adaptability	1	2	1	3	3
7. Maintainability	1	2	1	3	4
8. Low Cost	1	4	3	2	1
WEIGHTED SCORE		.52	.46	.70	.91

Figure 4-5. Rank Order Evaluation of Test Methodologies  
(equal weighting of design goals)

DESIGN GOAL (Section 4.6)	RELATIVE WEIGHT	METHOD (Section 4.7.1)			
		I	II	III	IV
1. Completeness	1	1	1	2	3
2. Context	1	1	1	2	3
3. Ease of Use	1	2	2	1	3
4. Expandability	1	1	1	2	3
5. Validation	1	1	2	3	3
6. Adaptability	1	2	1	3	3
7. Maintainability	1	2	1	3	4
8. Low Cost	10	4	3	2	1
WEIGHTED SCORE		.77	.61	.59	.56

Figure 4-6. Rank Order Evaluation of Test Methodologies  
(high weighting for cost)



#### 4.7.2.5 Evaluation of Test Validity

Method I is validated in the sense that independent Air Force and industry representatives have cooperated to test their individual interpretations of MIL-STD-1750 as described in Section 5.8.3. The level of validation proposed for Method II would add another vote to that process, in the sense of utilizing another independently developed simulator as an independent functional test. Method II is therefore at a higher rank than Method I. Likewise, Methods III and IV will undergo testing on an independently developed simulator, but will have the additional advantage over Method II of having another independent vote in the form of the test programs themselves and the test data generation procedures (cf. Section 5.8.2).

#### 4.7.2.6 Evaluation of Test Adaptability

Methods I and II utilize a large number of program modules organized by opcode. Each opcode test module consists of two to five parts each of which samples basic operation of the instruction, lack of extraneous interrupts, correct status word settings, correct interrupt generation, and correct operation of indexing. (cf. Appendix F). Typical opcode test modules require two hundred source lines to implement these five parts; source lines include a mixture of program code, test data, and expected results.

In contrast, Methods III and IV utilize program loops common to several instructions. These loops typically consist of sixty source lines of code. Test data and expected results are stored in tables separated from these code loops (cf. Sections 5.6.3 through 5.7.3).

Methods III and IV are considered more adaptable than Methods I and II because changes will normally be isolated to specific

code or data segments common to all instructions utilizing the changed feature. For example, expected results for all tests of integer divide by zero with 32 - bit result would reside in a single results table for Methods III and IV, but would appear in five different modules for Methods I and II. Another example of test adaptability would be instruction format changes such as modifying the definition of base registers for the B and BX formats. Methods I and II would require many modifications to be made in each of thirty-two test modules (about 6400 lines of code) while Methods III and IV would require modifications to a small number of instructions in each of 16 program loops (about 960 lines of code). Methods III and IV are clearly more adaptable because the code is functionally organized in several dimensions, not just opcode. Method I has been given a higher score than Method II because no modification of validation code (the simulator) would be required.

#### 4.7.2.7 Evaluation of Test Maintainability

All four certification test methods outlined will undoubtedly utilize equivalent program storage and maintenance techniques provided by the VAX-11/780 computer system. Method IV has the unique advantage of being able to archive procedures, test data, and certification test results in addition to having simple storage capabilities (cf. Section 5.2). Method IV is given the highest rating because of this advantage. Method III is rated higher than Methods I and II because it utilizes functionally

organized code segments than are common to several instructions rather than individually coding each opcode variant and data point to be tested. Method I is more maintainable than Method II because no simulator maintenance is required.

#### 4.7.2.8 Evaluation of Test Costs

The final detailed evaluation to be performed is that of cost. The lowest cost should receive the highest rank in this case. Obviously, Method I is the lowest cost alternative because it requires simple incorporation of available source programs into a VAX-11/780 resident data base. Recurring costs should consist of normal program maintenance, conversion of object code to the appropriate bootstrap media format, and operation of this very short test. Method II should be ranked next because it requires development of the fewest additional capabilities. Major additional costs for Method II are the increased maintenance costs for additional software (the simulator), and a significant development cost relative to costs for Method I in the form of testing requirements. Method III contains significantly more development than Method II, and therefore is ranked lower.

Any savings in maintenance costs for Method III over methods I and II are probably offset by increased program size (cf. Section 4.7.2.6). Method IV has development costs which are about 30% higher than for Method III (cf. Section 5.13 and Figure 5-9). In time, these additional costs will probably be covered relative to Method III, through greater ease of operation. Nevertheless, Method IV is ranked lower to indicate costs relative to the first unit tested rather than extending development costs over the life of the certification facility.

#### 4.7.3 Summary of Results

Figure 4-5 and 4-6 summarize the results of the detailed evaluation performed in Section 4.7.2. The scores shown in Figure 4-5 are for equal weighting of all seven design goals and for cost. The scores shown in Figure 4-6 are for high weighting of cost. Note that cost factors play a significant role in determining which methodology is chosen. In general, Method IV is technically superior to the other methods while Method I is lowest cost and will have the highest score whenever the relative weight of the cost goal is greater than 42% of the sum of the weights. The major decision to be made is between cost and technology!

## 5. RECOMMENDED APPROACH

This section recommends specific procedures, test designs, and support tools to be used by SEAFAC for certification of various implementations of MIL-STD-1750.

### 5.1 Overview of Recommended Approach

The recommended approach is designed to maximize the number of instruction codes and instruction code sequences to be executed by the unit under test while at the same time minimizing the amount of effort required to generate, document, and maintain test programs and data. Functional test modules are defined so that testing effort can be concentrated in a particular architectural entity. Most test modules have provisions for extending the test using random variables. Expected results are obtained from a simulator which is used as a secondary standard. This simulator will undergo rigorous testing during the generation of test programs. Test programs, data, and expected results are transferred between the unit under test and the VAX-11 test control computer thru a vendor supplied adapter unit which connects the UUT to the test control computer via a MIL-STD-1553B, RS-232C, or other agreed upon interface. A UUT resident program interfaces I/O device handler software provided by the vendor to the test modules used for certification. Summary reports are produced by the VAX resident test control program. These reports indicate which test modules were executed and provide descriptive information about the nature of any test failed. Provisions for identifying specific options that were/were not tested are automatically invoked by the test control program. Figure 5-1 shows the recommended certification facility.

### 5.2 Test Control Program

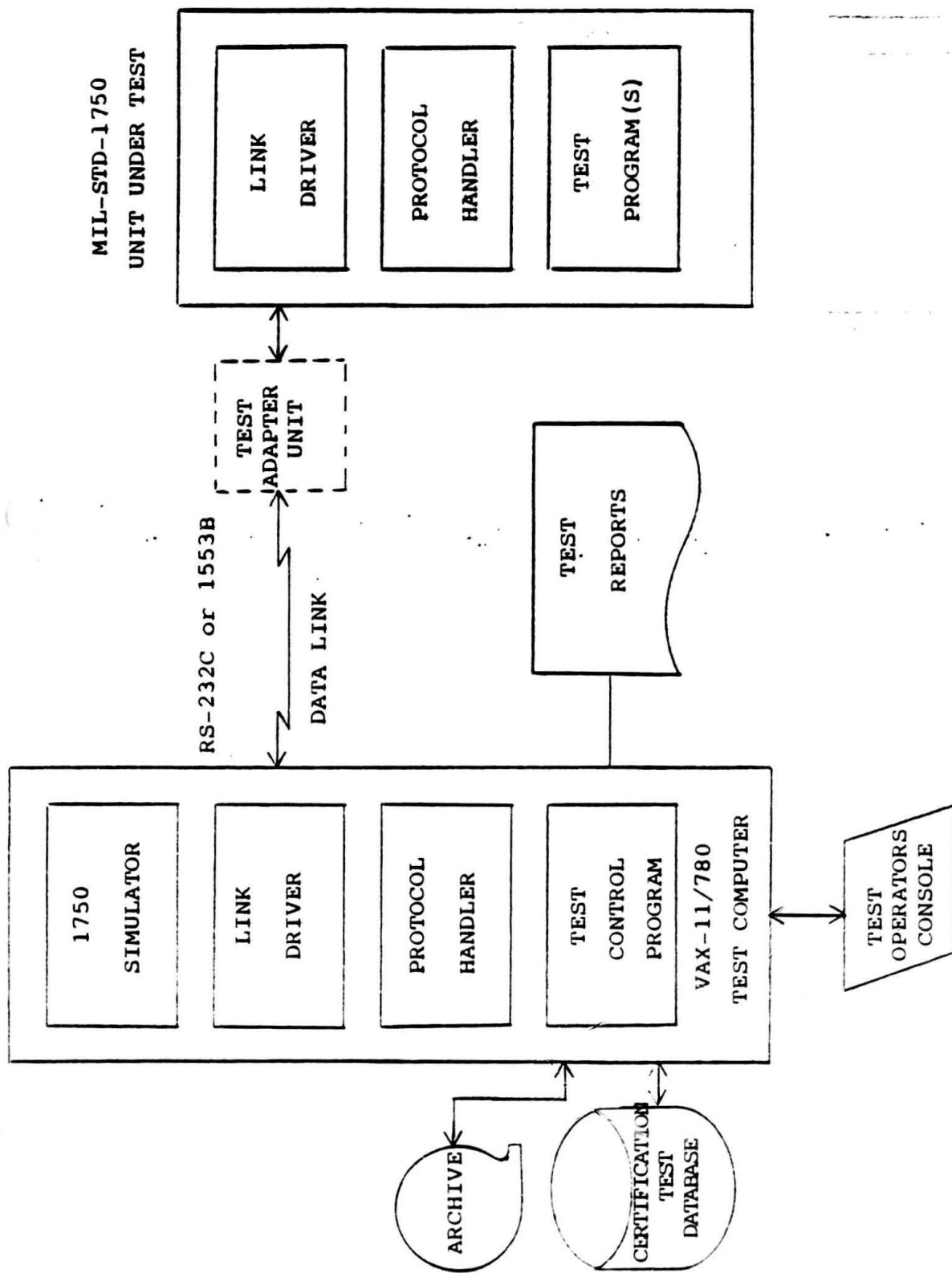


Figure 5.1. MIL-STD-1750 Certification Facility

A Test Control Program (TCP) capable of sequencing all phases of the certification process assures proper operation and application of test procedures by minimizing human intervention. The TCP utilizes program modules and data stored in the certification test data base to thoroughly exercise all UUT functions defined in MIL-STD-1750 and produce a summary report for each module exercised.

Simple operator controls allow the following functions to be performed.

- . Sequence thru all certification tests
- . Define the test sequence
- . Select a specific test module for execution
- . Add a new test module to the data base
- . Delete an existing test module from the data base
- . Archive an entire data base
- . Restore an archived data base
- . Save the UUT state (including memory) in the data base
- . Select the simulator or UUT for execution
- . Control/suppress levels of reporting
- . Control sequencing in event of errors
- . Provide diagnostic/debug aids

### 5.3 Protocol Handler

A Protocol Handler is provided as a means of allowing the test control program (TCP) to control the testing to be performed within the unit under test. The protocol handler is distributed between the test computer and the unit under test. Within the test computer, the protocol handler encodes/decodes message blocks to be transferred between the TCP and the unit under test. These message blocks are translated into appropriate action by

the protocol handler program residing within the unit under test.

#### 5.3.1 VAX Resident Protocol Handler

The portion of the protocol handler resident in the VAX-11 is designed to interface the test control program to the unit under test via a series of FORTRAN calls within the TCP. These calls provide the following functions:

- . Stop execution of any test module currently being executed
- . Restart or start execution of a test module
- . Transfer the contents of specified register(s) to the VAX
- . Transfer data to the specified register(s)
- . Transfer data to the specified memory location(s). (This function overrides any memory protection features.)
- . Perform the single instruction specified
- . Signal the specified interrupt(s)

A more detailed discussion of the VAX resident protocol handler is provided in Appendix E.

#### 5.3.2 UUT Resident Protocol Handler

The protocol handler resident in the unit under test is responsible for processing control information from the VAX computer and providing appropriate status messages to the VAX. This handler contains a storage area for the contents of the machine state. In effect, the 1750 resident protocol handler is responsible for switching the UUT between the test and control states. In the test state, the 1750 executes the currently resident test module using the hardware registers. When an



interrupt is generated in the 1750 in response to a message sent by the test control program, the contents of the hardware registers are saved in the machine state storage area and the message is processed by the protocol handler.

Individual messages may command that data be transferred between the test computer and the UUT memory or simulated registers. Provisions are made to release memory lockout when access is requested by the test computer. Once the requested message has been initiated, the machine state stored in the simulated registers is loaded into the hardware registers and processing of test code continues. Concurrent execution of test code and I/O transfer may be performed within the UUT provided memory lockouts were not removed and simulated registers are not affected.

Test modules resident in the UUT are able to invoke status transfer subroutines within the protocol handler. This entry point makes it possible for test modules to pass diagnostic information to the test computer for error reporting. Another entry point is available to return a complete status to the test computer and suspend execution of the test program.

#### 5.4 Data Link Drivers

Data link drivers provide a media dependent path between the test computer resident protocol handler and the protocol handler that resides in the UUT. These handlers transmit the received binary data blocks via a MIL-STD-1553B data bus, RS-232C communications link, or other agreed upon interface. Only those handlers actually needed to establish and maintain communications between the VAX and the UUT need to be configured at the time a certification is performed. For example, if a 1553B data link is to be used to certify a particular 1750 implementation, then neither the test computer nor the unit

under test is required to support an RS-232C connection.

The data link drivers are required to transmit and receive physical blocks of binary data without regard for specific data content. To accomplish this, different handlers are provided for RS-232C and 1553B links. In addition, the requirements for the VAX resident link driver may be different from the UUT resident handler.

#### 5.4.1 RS-232C Data Link Drivers

Data link drivers for RS-232C data links will require internal buffers, coding algorithms, decoding algorithms, synchronization, and error checking routines to establish a robust connection between the test computer and the UUT. Entry points into the data link drivers will be limited to transmit, receive, and check status functions.

The transmit routine will pass a buffer address and word count to the data link drivers and return a normal completion or error status to the calling program. The receive routine will pass a buffer address and maximum word count to the driver program which will return an actual word count or error status. A status checking function will allow the calling program to determine how many data words are currently buffered in the link driver or indicate that a data loss has occurred.

The binary data to be transmitted over the serial data link is converted to hexadecimal ASCII characters before transmission to avoid invoking any special functions that may be incorporated in the transmission of data path. In addition, synchronization, word count, and block check characters are added so that proper receipt of data can be verified. Each block of binary data transmitted on the RS-232 interface will consist of one or more space codes used for synchronization followed by

four ASCII characters indicating the 16-bit word count in hexadecimal, followed by ASCII characters representing the words to be transmitted in hexadecimal, followed by four ASCII characters representing the 16 low order bits of the twos complement sum of all the binary data transmitted including the word count.

The receiving data link driver responds to the receipt of a message block by sending a status word consisting of one or more space codes followed by the ASCII coded characters ACK, indicating correct receipt of data or NAK, indicating that an error was detected. The transmitting link will attempt to retransmit a message up to five times before an error status is returned to the calling program.

Data link drivers contain one or more fixed size buffers for receipt of unsolicited messages. The size and/or number of these buffers as well as the largest physical record that will be transmitted on the data link are defined by compile time variables that must be compatible within the drivers at both ends of the line.

#### 5.4.2 MIL-STD-1553B Data Link Drivers

Data link drivers for 1553B data links will require internal buffers for receipt of unsolicited messages; however, coding and decoding of the binary data is not necessary since the data link allows for transmission of binary data. Another advantage of the 1553B interface is that synchronization, control, error detection, and status reporting are all defined by the standard. The main responsibility of 1553B data link drivers is to provide a terminal address and divide the buffer provided by the calling program into block of 32 or fewer words as required by the 1553 protocol. The 1553B data link driver resident in the VAX-11 computer will act as the bus

controller. The UUT will act as a remote terminal which responds only to its own terminal address which may be selected by a compile time parameter. No action is required in response to any other terminal address including the "broadcast" address. Messages that are received in error will be retransmitted up to five times before an error status is indicated at the transmitting end.

### 5.5 MIL-STD-1750 Simulator

A MIL-STD-1750 instruction set simulator is provided for generating expected results and debugging test program code. The simulator is a set of FORTRAN subroutines that simulate all architecturally relevant features of MIL-STD-1750 in a manner that makes the simulator indistinguishable from a unit under test.

A software switch is provided within the VAX resident protocol handler to transfer control information and data to the simulator rather than sending it on to the device handler. In effect, the simulator is a unit under test that shares its device handler interfaces with the test computer protocol handler at one end, and the UUT protocol handler written in 1750 instruction set on the other hand. The simulator is "started" whenever the test computer protocol handler sends a message to the UUT, and execution is suspended whenever the UUT replies through its protocol handler.

The simulator is coded in such a way that it can be linked to programs other than the TCP by calls to the FORTRAN subroutines shown in Appendix C. These subroutines allow the retrieval of expected results and provide a means of generating and validating arbitrary sequences of code as described in section 5.6.3.

## 5.6 MIL-STD-1750 Test Programs

This section recommends a set of test programs to be executed within the unit under test to verify that it conforms to the requirements of MIL-STD-1750. These tests are functionally organized and should normally be executed in the order specified.

### 5.6.1 Factory Acceptance Test

Each vendor of a MIL-STD-1750 implementation should be encouraged to employ any hardware and/or software diagnostic tests they may have available to verify that the implementation to be certified conforms in every respect to the vendors specifications and expectations. This optional test would be most suitably employed while the UUT is in place at SEAFAC so that the likelihood of attempting to certify an improperly installed or somehow subtly damaged unit is minimized.

### 5.6.2 Data Link Test

A test of the data link software should precede the actual certification tests to be employed. This test is designed to verify that data can be successfully transferred between the test computer and the UUT. It utilizes the protocol functions described in Appendix E to perform the following operations.

- . Transfer a large number of data words to UUT memory
- . Retrieve the same number of data words from UUT memory and verify contents
- . Transfer and retrieve successively smaller data blocks assuring that data is not added or deleted
- . Test the transfer of all 65K possible bit patterns

### 5.6.3 Instruction Tests

A series of test modules are required to thoroughly test all instruction codes. Several test modules are needed since "known good" values stored within the UUT for comparison to derived values requires more than 65K of data storage. The "known good" values include the minimum required operands described in section 3.4.1 plus an arbitrary number of randomly generated operands that are produced as described in section 5.7.

Figure 5-2 shows a control structure for the Instruction Tests which derives information about the data to be tested from a series of tables. An opcode table contains the opcode to be tested, a type code that indicates the test loop to utilize for testing the instruction, and pointers to UUT resident tables that contain expected results and operands used to generate the results. Figure 5-3 shows a typical test loop that provides for testing of all combinations of RA and RX registers for the instruction under test with a specifiable number of operands to be used in the RA and RX fields, and a (possibly) different set of operands stored in memory.

Use of tables to define operand values and expected results offer many distinct advantages:

- . New or different operands can be added at will up to the limits imposed by available memory
- . Operands and instructions can be independently repositioned in memory by use of separate compiles and/or relinking
- . Program code can remain in UUT memory while new data tables are transferred
- . Simple coding sequences within the VAX can divide available data space into two or more partitions.

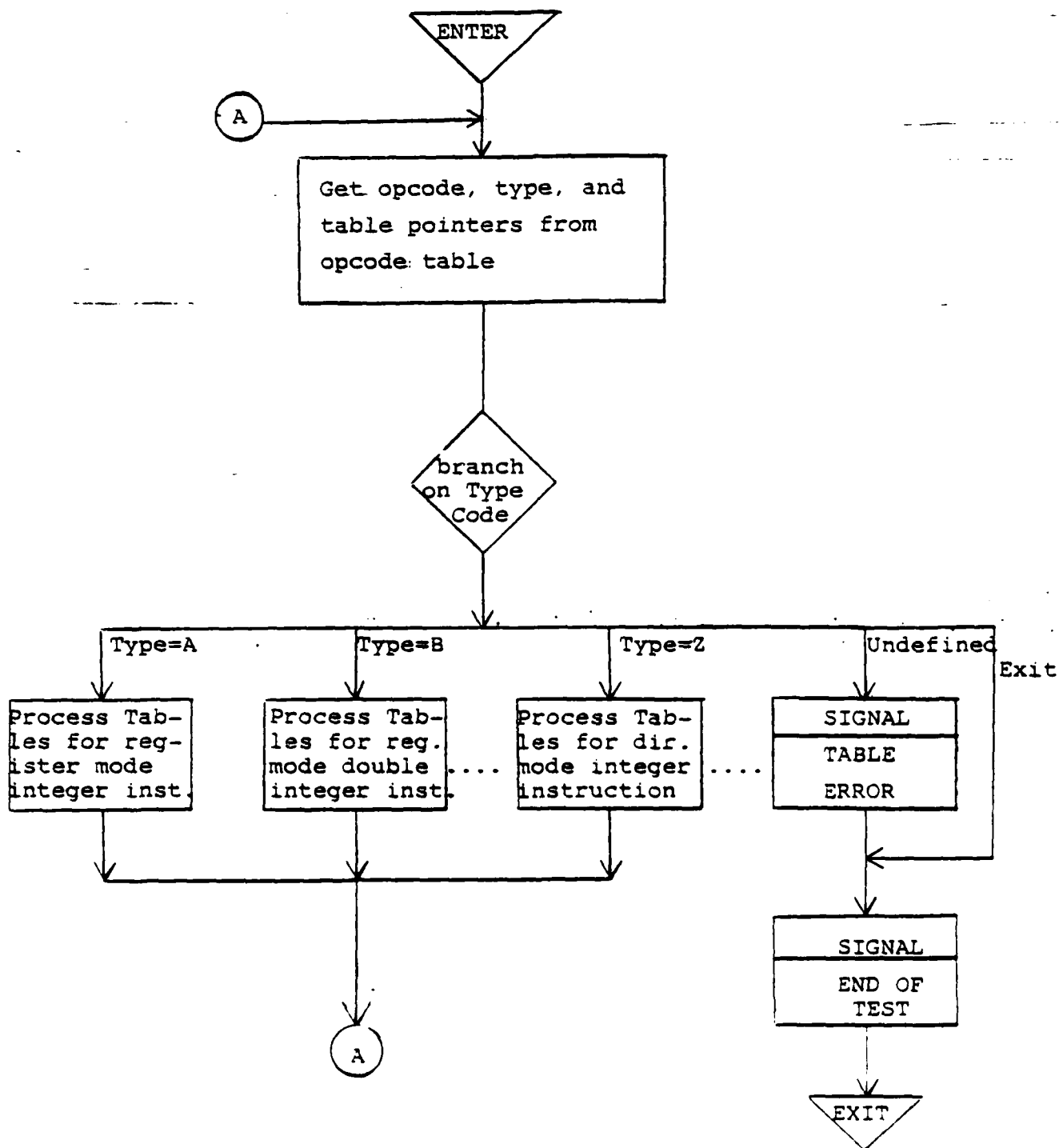
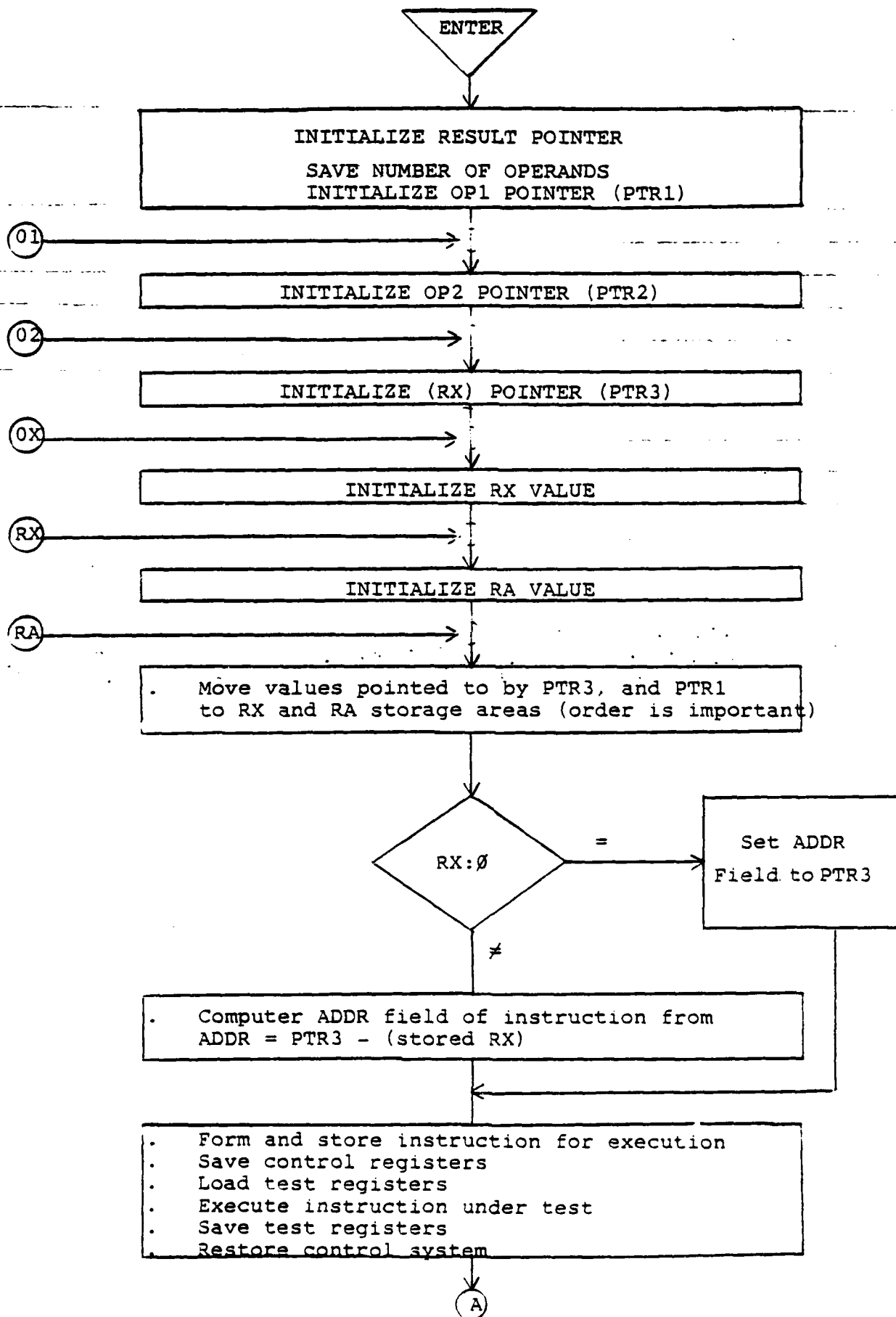


Figure 5.2 . Instruction Test Control Structure





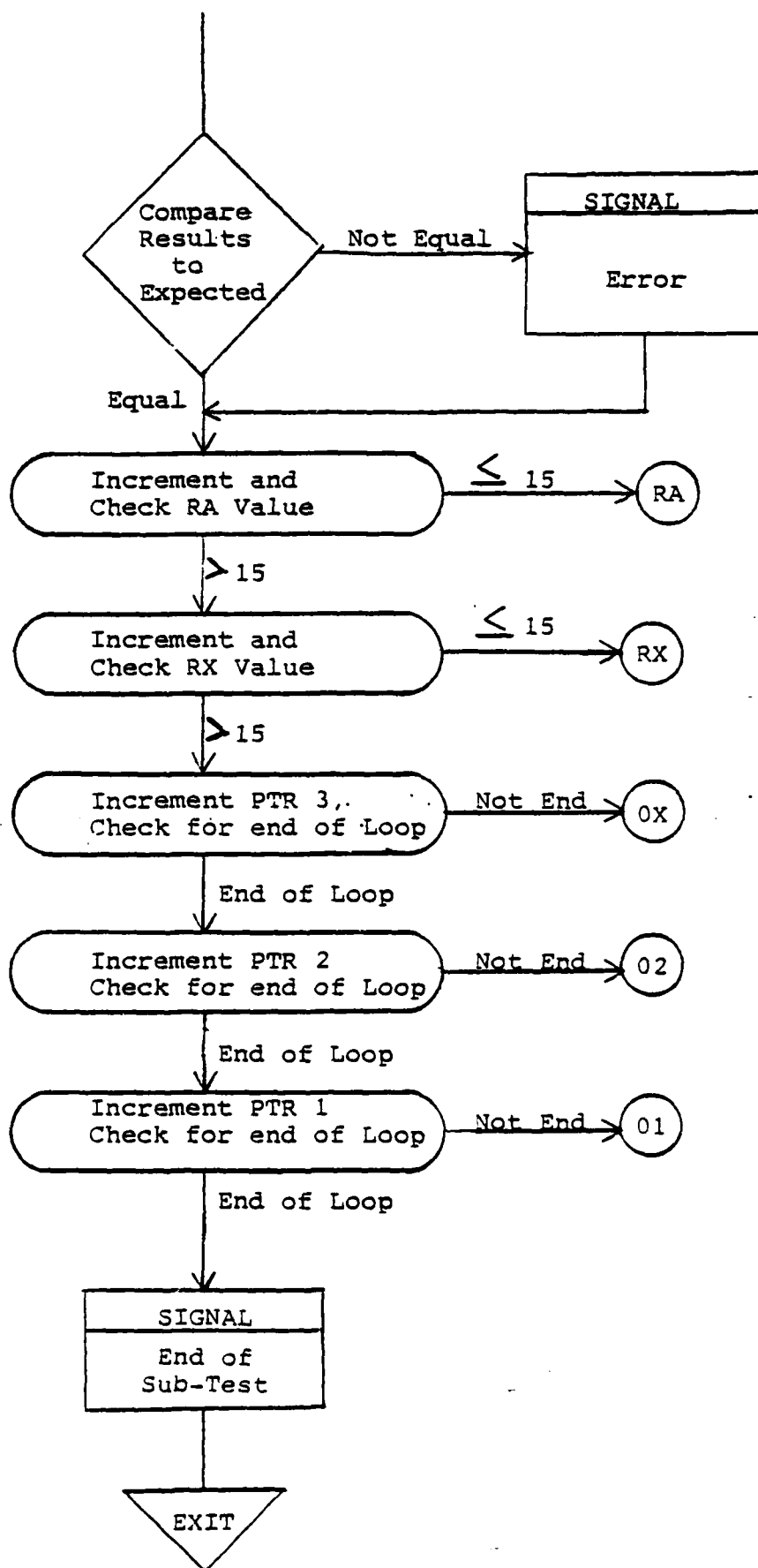


FIGURE 5.3. (CONTINUED)

One partition can be loaded from the VAX while the other is being processed by the UUT. This allows the test to proceed while interrupts and concurrent I/O are being processed.

Several points should be highlighted at this time. First, not all instructions can be effectively tested in this mode of operation. The MOV instruction, for example, has a unique format and requires that one or more interrupts occur during the actual test of the instruction. To thoroughly test the MOV instruction requires the injection of an interrupt that is not synchronized with the MOV instruction itself, and the interrupt processing routine must be able to assist in verifying that the interrupt occurred during the operation and not immediately before or after instruction execution. The level of interaction between the test computer and the UUT required for this type of test warrants a separate test component.

A second point to be highlighted is that the results that should be checked include interrupts, status word contents, and affected memory contents, not just the contents of affected registers. Detailed study of each instruction to be performed is required in order to select the appropriate test data, expected interrupt conditions, and expected status word contents; these factors will determine which test type is appropriate for the instruction.

A third point is that information needed for detailed error reporting as defined in section 3.7.3 should be provided at the time an error condition is indicated to the test control program even if this information is not used for automated reporting. One mechanism for providing this information is to pass the address of a data record containing detailed error information to the test control program as a status indicator. This record can be retrieved by the TCP if detailed reporting

is required; otherwise the test procedure can continue or terminate at the discretion of the TCP and/or test operator.

#### 5.6.4 Memory

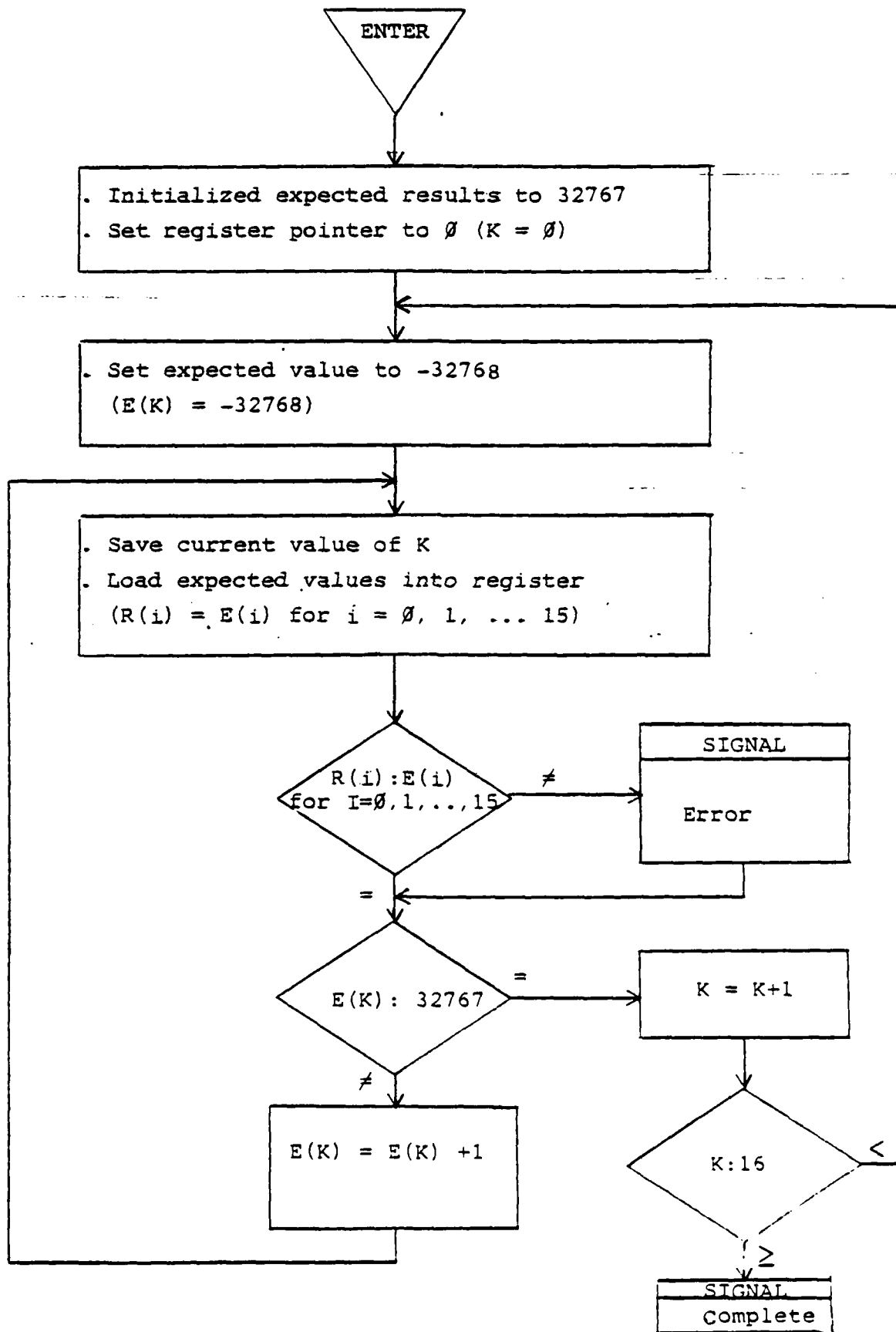
The AFAL ATP memory test program described in Appendix F should be modified to allow its incorporation into the test program(s) (cf. Section 2.4.2) by protecting the UUT resident protocol handler and device handler areas from access. Tests of optional features such as memory protect and start-up ROM should be removed from the memory test and placed into separate test modules that can be included (excluded) based on the configuration of the UUT. Diagnostic features of the test such as restricting the patterns to be tested should be removed or controlled thru interaction with the TCP to avoid "short-circuiting" of the entire test procedure during the certification process.

#### 5.6.5 Register Tests

A test of all general registers is recommended (cf. Section 2.4.2). This test should check for all 64K data patterns in each of the 16 general registers. Tests for correct contents of all 16 registers should be made after each test pattern is generated. Figure 5-4 shows the register test component. Contents of other registers should be checked in the I/O test.

#### 5.6.6 Derived Address Tests

Although the Instruction Tests as defined earlier will test the ability of each instruction to access data in a limited number of the possible derived addresses, a separate test of derived address calculations is recommended. It will provide a systematic and more comprehensive check on the ability of the UUT to correctly compute derived addresses on each addressing mode. A summary of the fourteen distinct addressing



<u>Addressing Mode</u>	<u>Derived Address</u>	<u>Derived Operand</u>
1. Register Direct	RB	(RB)
2. Memory Direct	ADDR	[ADDR]
3. Memory Direct Indexed	ADDR+(RX)	[ADDR+(RX)]
4. Memory Indirect	[ADDR]	[[ADDR]]
5. Memory Indirect Indexed	[ADDR+(RX)]	[[ADDR+(RX)]]
6. Immediate Long	-	I
7. Immediate Long Indexible	-	I+(RX)
8. - with RX=0	-	I
9. Immediate Short Positive	-	+I+1
10. Immediate Short Negative	-	-I-1
11. IC Relative	DSPL+(IC)	-
12. Base Relative	DSPL+(BR)	[DSPL+(BR)]
13. Base Relative Indexible	(BR)+(RX)	[(BR)+(RX)]
14. - with RX=0	(BR)	[(BR)]

Note: Double Precision and Floating Point Instructions reference DA, DA+1 as a unit, and Extended Floating Point Instructions reference DA, DA+1, DA+2. The notation used here is defined in MIL-STD-1750.

Figure 5-5. Addressing Modes in the 1750 Instruction Set

modes defined by MIL-STD-1750 is given in Figure 5-5. Note that for the Immediate modes 6-10, the derived address is not relevant because it is part of the instruction itself. Thus no addressing tests need be performed for the Immediate modes separate from the Instruction Tests. The IC Relative mode 11 is only used for the Branch instruction, which is adequately tested already in the Instruction Tests. In addition, the Register Direct mode 1 is fully tested in the Register Tests, which verify both data values and derived address values for all registers. This leaves seven addressing modes to test: 2-5 and 12-14.

Since the Memory Test and the Register Tests are concerned with the storage and retrieval of possible data values, this addressing test need not be concerned with any variety in data values manipulated except so far as they help verify that the appropriate address was indeed reached (e.g. using addresses as the data). Also, since the Instruction Tests will cover all OPCODESs, this test can be constructed using a single instruction such as LOAD, which spans all addressing modes. The test strategy then is to reference all derived addresses of the mode being verified and confirm the reference using instructions which are of a different addressing mode. For memory direct, this would amount to performing LOAD's for all 64K addresses, pre-storing and confirming address data with instructions of some other addressing mode.

For derived addresses which are indirect or involve a sum of a register and an address, displacement, or some other register, the number of possible addresses is too large to be testable in a reasonable time. For example, with Memory Direct Indexed mode the complete specification of ways to compute ADDR+(RX) is 64K x 64K x 16, counting all addresses, all values of RX, and all RX's. A reasonable compromise with completeness is to perform three subtests, fixing two of the three variables

each time while varying the third over its entire range. The values which are fixed would be chosen at random from their range of possibilities. This scheme is easily extended to apply to the other modes which involve sums in computing their derived addresses.

#### 5.6.7 Input/Output and Interrupt Tests

A functional components for testing all non-optional I/O capabilities is recommended. These tests should confirm correct addressing and value storage for all registers available through input/output instructions. These registers include:

- . Status Word Register (SW)
- . Fault Register (FT)
- . Interrupt Mask (IM)
- . Pending Interrupt (PI)

Functional testing of these registers requires careful ordering of instructions, use of specialized instructions, and in some cases use of instructions that will be added in the first revision of MIL-STD-1750. For example, testing of the status word register will require that all combinations of the four upper bits be tested. The standard does not currently define the use or operation of the lower twelve bits of the status word; therefore, the results of testing should be reported, but strictly speaking, any value that is returned is allowed. (This particular problem and many others will be addressed in MIL-STD-1750A.)

Interrupt testing can logically be done in this test component although interrupts normally associated with overflow, underflow, illegal opcode, and other errors that can be generated by software will be more extensively tested in other test modules as well. Additional capabilities are being defined

in MIL-STD-1750A that will allow more complete testing of fault detection and interrupts. The ability to set the contents of the pending interrupt register is crucial to testing interrupt acceptance and priority without introducing implementation-dependent test hardware.

#### 5.6.8 Jump and Branch Tests

Proper operation of conditional and unconditional branching should be performed in a separate test module. Conditional branches that do not cause control to be transferred should be checked before those that do cause control to be transferred as shown in Figure 5-6. Conditional jumps and branches will require 256 jump instructions to be executed for each of the five addressing modes (D,DX,I,IX, and ICR) as shown in Figure 5-7.

Several jumps require special handling by individual tests. For example, the stack pointer and return address should be verified for the stack IC and jump to subroutine (SJS) instruction.

#### 5.6.9 Random Instruction Sequence Tests

A series of test modules that incorporate sequences of instructions should be provided. These sequences should be of two forms. The first form would consist of a series of executions of the same instruction. The purpose of this type of test is to attempt to fill any pipeline(s) that may have been used to implement a particular function. The number of instructions in the sequence should be large enough to fill any reasonable size pipeline. One-hundred instructions would be more than adequate. One convenient method for generating this sequence of instructions is to modify the loop shown in Figure 5-3 to store copies of the same instruction in successive memory



STATUS WORD	CONDITION CODE		BXX		BLT		BEZ		BLE		BGT		BNZ		BGE		BXX	1000		1010		1100		1110		1	
			0000		0001		0010		0011		0100		0101		0110		111										
			0		1	2	3	4	5	6	7	8	9	A	B	C	D	E	F								
0000	0		N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
0001	1		N	J	N	J	J	N	J	N	J	N	J	N	J	N	J	N	J	N	J	N	J	N	J	N	J
0010	2		N	N	J	J	J	N	N	J	J	N	N	J	J	N	N	N	N	J	J	N	N	J	J	J	J
0011	3		N	J	J	J	J	N	J	J	J	N	J	J	J	N	J	N	J	J	J	N	J	J	J	J	J
0100	4		N	N	N	N	N	J	J	J	J	N	N	J	J	N	N	N	N	N	J	J	N	J	J	J	J
0101	5		N	J	N	J	J	J	J	J	J	N	J	N	J	J	J	J	N	N	J	J	J	J	J	J	J
0110	6		N	N	J	J	J	J	J	J	J	N	N	J	J	J	J	N	J	J	J	J	J	J	J	J	J
0111	7		N	J	J	J	J	J	J	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J
1000	8		N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	J	J	J	J	J	J	J	J
1001	9		N	J	N	J	J	N	J	N	J	N	J	N	J	N	J	J	N	J	J	J	J	J	J	J	J
1010	A		N	N	J	J	J	N	N	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J
1011	B		N	J	J	J	J	N	J	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J
1100	C		N	N	N	N	N	J	J	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J
1101	D		N	J	N	J	J	J	J	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J
1110	E		N	N	J	J	J	J	J	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J
1111	F		N	J	J	J	J	J	J	J	J	N	J	J	J	J	J	N	J	J	J	J	J	J	J	J	J

J = Jump should be taken

N = Next instruction should be executed

Figure 5.7. Conditional Branch/Jump Matrix



locations. If this technique is used, precautions must be taken to avoid problems associated with modification of the derived address of the pre-stored instructions. This occurs in the DX format when  $RA=RX$ . Simple checks can be made to bypass tests of this type.

Final results for this sequence test are similar to those encountered in the single instruction tests except that the number of interrupts becomes significant, and more than one type of interrupt may occur. This type of testing may not be valid for certain combinations of instructions and operands. For example, the single precision integer divide instruction in register format (DVR) will yield indeterminant results when  $RB = RA + 1$  and the remainder of a divide preceeding the last in the sequence is zero. Subtle conditions such as this can only be derived from a detailed analysis of individual instructions as has been indicated in Section 5.6.3. Fortunately, in the case of integer divide the integer overflow interrupt can be used to indicate that the results are not well defined.

A second form of instruction sequence test provides random data and instruction sequences (cf. Section 3.5.2) by utilizing special features of the MIL-STD-1750 simulator. The following procedures indicate the method to be used to generate a random set of code and data using the simulator.

1. Define a memory space to be used for the test.
2. Fill the simulated memory space with random number sequences. (These sequences may have specified statistics).
3. Initialize simulated general registers with random number sequences.
4. Select a subset of simulated memory to be designated as the instruction area.
5. Load the IC register with a random number whose value within the instruction area.

6. Set breakpoints to halt execution of the simulator if any memory reference outside the defined memory space is attempted.
7. Set breakpoints to halt execution of the simulator if any execution reference outside the instruction area is attempted.
8. Set breakpoint to halt execution of the simulator if any operand references are attempted inside the instruction area.
9. Save a copy of the initial machine state including memory contents for later reference.
10. Initiate execution of a single instruction.
11. Examine the completion status of the simulator. If no breakpoint conditions were satisfied, repeat step 10 otherwise go to the next step.
12. One of the test ground rules has been violated; therefore the instruction pointed to by the IC should be modified by selecting a new random variable to replace the instruction in the copy saved in step 9.
13. The initial machine state and modified memory contents are loaded from the saved copy and step 10 is repeated.
14. Step 10 should be repeated N times. A jump to the test completion entry in the UUT Protocol handler should be inserted into the stored code at the IC location on the (N+1)st instruction execution.
15. The modified copy of the initial machine state should be reloaded and a maximum of  $N + 100$  instructions should be simulated.
16. If  $N + 100$  operations are executed without a jump to the test completion entry, the attempt should be aborted. Another attempt to create a test can be made by starting at step 1.
17. If the test completion entry is reached, the final machine state including memory contents are saved for comparison with the results produced by the UUT.

The results of this long and arduous (but fully automated) process is a test program (the initial machine state), and the expected results (the final machine state). A specific unit under test can be loaded with the stored test code and the test initiated under control of the test control program (TCP). When the UUT indicates test completion, the TCP retrieves the current machine state from the UUT and compares these results to the expected results stored in the test database.

#### 5.6.10 Context Tests

Application programs provide a natural extension to the available test software. Any well-documented program or subprogram that does not depend upon time or implementation specific facilities can be used to increase the level of test completeness. These programs have significant advantages over the random sequences of code described in the previous section in that they represent a realistic mix of instructions that serve a useful purpose (cf. Section 3.5.2). Extensive use of these routines would provide proof positive that these specific programs can be used with certified implementation of 1750.

Availability of these mission software components will become increasingly available with the passage of time, but few will be available at the time of the first 1750 certification. One acceptable first substitute for these components are the DAIS and Sperry Univac ATP's for the AN/AYK-15A. These tests should be augmented as mission software components become available.

Among the first general purpose software components that will become available will undoubtedly be mathematical subroutines such as SIN and COS functions. These functions can be very easily tested by using the well known trigonometric identity:

$$\text{SIN}^2 (x) + \text{COS}^2 (x) = 1$$

A simple test procedure would perform the necessary operations for a wide range of values for (x) and assure that the result was within the limits of accuracy of the algorithm implemented on the 1750. Use of the 1750 simulator or other secondary standard to generate results would allow testing for exact values rather than using extreme limits.

## 5.7 Test Data Generation

The tables of opcodes, operands, and expected results described in Section 5.6.3 can be built by use of assembly directives, by use of VAX resident data generation programs, and by use of simulator results. All three methods of generation are desirable for use in validating the test facilities as described in Section 5.8. Any one method of test data generation is adequate for generating the appropriate tables; alternate methods provide an independent means of verifying the results.

The type code described in Section 5.6.3 is fixed for each instruction. The actual code to be assigned is dependent upon the particular implementation used for the instruction test. One organization of the instruction test would utilize a sorted list of all instructions similar to Appendix F to define when a type code is needed. This list is sorted in the order: addressing mode, operand/result format(s), and instruction class (load, store, register, etc). Each change in the contents of one of these fields would indicate need for another type code. Once the relationship between type code and operation code has been established, test tables can be generated using any of the following techniques.

### 5.7.1 Test Data Generation Using the Cross Assembler.

Tables of test data can be generated using the capabilities of the VAX hosted 1750 cross assembler. Assembly-time capabilities play a vital role in determining the relative ease of generating these tables. Factors that influence these capabilities are:

- . The constant data formats supported (integer, double, floating point, etc.)
- . The operations that can be performed on each data format (add, subtract, multiply, divide, shift, logical, etc.)
- . The ability to utilize parameterized macros
- . The ability to utilize assembly-time tables
- . The ability to process assembly-time decisions

A section of assembly code capable of generating tables of operands and results is shown in Appendix G. This sample code indicates the relative simplicity of generating large sequences of test code when appropriate assembly-time facilities are available. The samples chosen are purposely small since the number of output cells for this particular example is approximately five times the square of the number of operand pairs entered. If one hundred operand pairs were entered, slightly over fifty thousand data values would be generated.

#### 5.7.2 Test Data Generation Programs

Relatively simple programs can be written in FORTRAN (or other suitable language) to generate data tables. These programs can be used to produce appropriate source or core images by accepting user supplied or randomly generated operands, performing the required functional operations, and producing a table of expected results using the internal data format(s) of the VAX. Utility routines for conversion of internal format data to equivalent 1750 data representation must be

provided.

### 5.7.3 Test Data Generation Using the 1750 Simulator

Test programs can be modified to act as test data generation routines by replacing the section of code that compares results with code that stores the computed results in the appropriate simulated memory cells. Operand tables can be either manually entered into the test program code, or generated in the test computer and transferred into the appropriate simulated memory area by invoking protocol routines. Once the operands and results are in simulated memory, they can be retrieved from simulated memory and stored in the test data base for later use with the unmodified test program.

## 5.8 Test Validation

Test programs and the 1750 simulator should be verified to be correct before beginning certification of a 1750 implementation. In effect, this test validation is the first certification of an implementation of MIL-STD-1750, the 1750 simulator. The following paragraphs outline procedures that can be used to validate both the test programs and the simulator. The validation described uses majority vote and is always subject to question. Confidence in the accuracy of the simulator and test programs used as secondary standards is relatively high because of the number of independent checks of correctness that are made.

### 5.8.1 Design and Validation of the Simulator and Test Program

Design and coding of the 1750 simulator and the 1750 test programs should be carried on independently until both are complete and ready to validate. This constraint is relatively



easy to assure because the language and implementation constraints are so different. The simulator will use MIL-STD-1750 to form the basis of a software design that will be implemented in FORTRAN operating on and targeted to a VAX-11 computer. The test programs on the other hand use MIL-STD-1750 as the basis of their design. They will be implemented entirely in 1750 assembly code. Note that each component attempts to use all opcodes and addressing capabilities of MIL-STD-1750 within its implementation. Therefore, by attempting to concurrently test the two components (assuming some initial level of independent code debugging including desk debugging) any code discrepancies can be resolved by reference to MIL-STD-1750.

Each program fault or error indication encountered during the test must be carefully analyzed to determine if the indication is caused by an error in the simulator design, the test code, or a difference in interpretation of MIL-STD-1750. A test review board consisting of one or more individuals responsible for simulator implementation, one or more individuals responsible for implementation of test code, and one or more representatives of SEAFAC should be responsible for giving final approval to all program changes during test validation. Any changes made should have unanimous approval of the review board. If unanimous approval cannot be arrived at, an ambiguity or discrepancy in MIL-STD-1750 will undoubtedly be involved. Testing of the disputed code will be bypassed, and the issue brought before the 1750 users group for resolution. Any certification to be performed would then note that testing in the disputed area would be limited to reporting results as described in Section 3.7.

#### 5.8.2 Data Generation for the Initial Test Program

Test data used for the purpose of validating the simulator and

test program should include the boundary conditions described in Section 3.6. This data should be manually generated and incorporated within the source code of the data tables described in Section 5.6.3. Manual methods of data generation would include use of calculators and small programs to generate expected results, but care should be taken not to use the same set of programs and programming assumptions used in coding the 1750 simulator. This restriction will assure that the simulator logic is not being tested against itself.

A minimum of two values for each operand should be chosen at random between each pair of boundary conditions. For integer and double precision data types a total of nine operand values would be used:

- . zero
- . minimum positive value (+1)
- . maximum positive value
- . minimum negative value
- . maximum negative value (-1)
- . two random positive numbers
- . two random negative numbers

These nine operand values would require a total of nine times nine or eighty-one "known good" values to be computed for each arithmetic type. Use of assembly-time tools such as those described in Section 5.7 will significantly reduce the drudgery involved in this effort.

Floating point operands will require significantly more patience to develop unless techniques similar to those described in Section 5.7.2 are utilized. The results of any data generated in this manner should be randomly sampled and verified by manual techniques before verification begins.

### 5.8.3 Independent Verification of the Test Validation

A cross check on the test and simulator validation procedure will be performed by executing the Automated Test Program (ATP) produced for the Air Force Avionics Laboratories (AFAL) DAIS program on the 1750 simulator. This program has undergone testing similar to that described in Section 5.8.1 except that a different simulator was first utilized for testing; then the program was executed on two MIL-STD-1750 implementations: one produced by Sperry Univac and the other produced by Westinghouse.

Any discrepancies noted in the results will be resolved by the same review board described in Section 5.8.1. Any discrepancies and possible resolutions of difference in the ATP will be reported to the DAIS program office; however, no mandatory action is required.

### 5.9 Reporting Test Results

The Test Control Program (TCP) will produce an automated report file which can be saved as a permanent and reproducible record of the certification results. The first portion of this file is a fixed format report containing the information described in Section 3.7.2. Descriptive information about the unit being certified is entered via the operator console in response to requests generated by the TCP. As each program block is transferred from the certification test data base, under control of the TCP, the start time for the test is entered into the report file, and the completion time is entered into the appropriate slot.

The TCP receives error information from the currently resident test module as described in Section 5.3.2. This error information is stored in an error diagnostic file, along

with information that indicates the test module, type of failure, expected results, and received results. This information can be processed by a separate report generation component to analyze the nature and extent of any failure. Results of operations that have undefined results (e.g. results of integer divide by zero) are stored in the diagnostic file but are listed separately from error diagnostics.

#### 5.10 Changes to MIL-STD-1750

MIL-STD-1750 is currently being revised to provide improved definition of existing requirements and to extend the defined capabilities to include extended memory and improved I/O. These enhancements will obviously require additional features to be added to the test components described in Section 5.6.

The cost impact of these changes will be minimal for the programs defined in Section 5 for three reasons. First, all changes to the MIL-STD can be incorporated during the detailed design phase without the need for re-programming. Second, costs may actually be reduced for some tests because ambiguous or difficult to interpret specifications are being revised in such a way that specific results are expected (e.g., results of divide by zero). Third, the recommended approach defines that functional testing of new features such as memory extension be tested in a manner that requires a linear extension of the amount of test code required rather than an extensive modification or addition as would be required if the existing ATP's were to be employed (cf. Sections 2.4.2, 4.4.1, 4.5, 4.7.2.6).

Future changes to MIL-STD-1750 are anticipated in the recommended approach. The 1750 simulator and the certification test programs will undoubtedly require extension to meet these changes as they develop. Each revision of the test data base, test control program, and 1750 simulator should be archived at the

DOCUMENT NO.

PX 13243

SPIERRE UNIVAC

time it is released so that the ability to certify or re-certify to a previous level of MIL-STD-1750 is maintained.

### 5.11 Distribution and Control of Certification Test Software

Widespread distribution of Certification Test code is recommended. This distribution of material will tend to assure continued use of the tests in development and checkout of both hardware and software components. If indeed the 1750 vendors extensively utilize the tests in the development of new implementations of 1750 and verification of continuing compatibility of previously certified designs, then additional confidence in the validated test components will be developed.

Test components should be distributed in core image format with appropriate source listings and/or as symbolic source images on magnetic tape. Components such as the test control program, protocol handlers, and device drivers will be provided on request but it is assumed that the vendor will not necessarily have a VAX computer available to control processing. This situation can be remedied by the vendor in either of two ways. First, the TCP, protocol handler, and 1750 simulator can be converted to an available test computer and appropriate device handlers supplied in both the test computer and the unit under test. Or second, the UUT resident protocol handler can be replaced by a simple control program that displays or stores results on a console device or storage media provided by the vendor. The test setup for the second alternative may be as simple as a maintenance console and bootstrap load device with stop instructions at the protocol entry points for signaling errors and completion.

SEAFAC should assume management responsibility for correcting errors or omissions in test code that may be detected by vendors that are under contract to develop or produce implementations of MIL-STD-1750. These errors or omissions should be indicated to SEAFAC in writing with sufficient detail to allow a determination of the validity of the claim. Any disputes that may arise from this process should be resolved by the MIL-STD-1750 User Group as defined in Section 5.1.1.

The purpose of distributing this test code is to allow each vendor to prepare for certification of a particular 1750 implementation. The certification process itself can only take place with a released version of the certification programs. These programs may include different data sets or program code from that provided to the vendor.

#### 5.12 System Resource Requirements

The specific resources required to implement, maintain, document, distribute, and control programs and test procedures defined in this recommendations section of the final report are listed in Figure 5.8. The hardware facilities available at SEAFAC are ample to support any software development and maintenance procedures needed for certification of the MIL-STD-1750 ISA. The only hardware additions that would be necessary for support of the MIL-STD-1750 certification procedures are the incorporation of additional types of media for program data transfer to a specific implementation of 1750 and/or the addition of a vendor supplied UUT adapter unit.

The software facilities available on the VAX-11/780 are ample to provide program maintenance and to develop VAX resident data generation and analysis capabilities that are described in Sections 5.1 thru 5.5 and Section 5.7. Missing support software capabilities include a 1750 cross assembler, a link-edit capability, and a 1750 ISA simulation capability. These software components are or soon will be available through other government facilities located in Dayton.

Currently available cross assembler and linkage capabilities include the ALAP assembler and LINKS linkage editor now being used by the DAIS project at AFAL, and the JOCIT facility at RADC. These capabilities are written in FORTRAN and are highly portable. Another FORTRAN based capability being

COMPONENT DESCRIPTION	HARDWARE/SOFTWARE	SOURCE(S)	AVAILABLE DATE
VAX-11/780 Computer	Hardware	SEAFAC	Now
Random Access Mass Storage	Hardware	SEAFAC	Now
Magnetic Tape Unit(s)	Hardware	SEAFAC	Now
Printer/Hardcopy Device	Hardware	SEAFAC	Now
Control Console	Hardware	SEAFAC	Now
RS-232 Capability	Hardware/Software	SEAFAC	Now
1553B Bus Capability	Hardware/Software	SEAFAC	June 1980
FORTRAN Computer (VAX)	Software	SEAFAC	Now
Linker (VAX)	Software	SEAFAC	Now
Assembler (VAX)	Software	SEAFAC	Now
Operating System (VAX)	Software	SEAFAC	Now
FORTRAN Random Access to Files	Software	SEAFAC	Now
1750 Cross Assembler	Software	AFAL (DAIS)	Now
		ASD (ADAM)	June 1980
		Other	TBD
1750 Macro Capability	Software	ASD (ADAM)	June 1980
		Other	TBD
1750 Link Edit	Software	AFAL (DAIS)	Now
		ASD (ADAM)	June 1980
		Other	TBD
1750 Simulator	Software	AFAL (DAIS)	May not be Trans- portable
		ASD (ADAM)	June 1980
		Other	TBD

FIGURE 5.8 . CERTIFICATION SYSTEM RESOURCES



developed for the ADAM program is available within ASD, and includes a macro generation capability that is highly desirable (see Section 5.7.1). A preliminary copy of the ASD cross-assembler and linkage capability has been installed on the PDP-11/55 computer at SEAFAC, but has not been thoroughly tested. A released VAX version of the ADAM support software package will be available in June of 1980. In short, the support software tools needed to provide a MIL-STD-1750 certification capability are readily available at low cost and low risk.

### 5.13 Summary of Resources to be Developed

The certification test components and capabilities to be developed are shown in Figure 5.9. Several of these items are unique to the implementation being certified and are therefore listed as vendor supplied items; all other items are to be supplied by the Air Force (SEAFAC) at the time certification is attempted. In general, item numbers twelve thru twenty-one are mandatory and cannot be eliminated or reduced in scope without affecting test completeness. Other items with the exception of the 1750 bootstrap capability may be considered as optional in that alternate methods can be developed to support the necessary documentation, archive, and test administration procedures with a corresponding need for greater overall knowledge and care in administering certification tests.

Estimates of the number of source lines required to generate a particular test component, the number of 1750 memory cells required to hold the resultant code, and relative development cost estimate were made by implementing a few typical sections of code and extrapolating from those code segments.

ITEM NO.	COMPONENT DESCRIPTION	HARDWARE/ SOFTWARE	AF/V	PARAGRAPH(S)	SOURCE LANGUAGE	SOURCE LINES	1750 CORE SIZE	RELATIVE COST
1	1750 Bootstrap Capability	Hardware/ Software	V	3.1.1				
2	Test Adapter Unit	Hardware	V	5.1				
3	UUT Link Driver	Software	V	5.4	1750 ASM	400	1000	4
4	Test Control Program	Software	AF	5.2	FORTAN	2000		40
5	Protocol Handler (VAX)	Software	AF	5.3.1	FORTAN	1000		5
6	Protocol Handler (UUT)	Software	AF	5.3.2	1750 ASM	600	1000	6
7	Link Driver (VAX-RS-232)	Software	AF	5.4.1	FORTAN	400		2
8	Link Driver (VAX-1553B)	Software	AF	5.4.2	FORTAN	300		2
9	FORTAN Interface to 1750 Simulator	Software	AF	5.5, 5.8.1	FORTAN	1000		5
10	Factory Acceptance Test	Software	V	5.6.1	1750 ASM			
11	Data Link Test	Software	AF	5.6.2	FORTAN	200		1
12	Instruction Test	Software	AF	5.6.3	1750 ASM	4000	6000	120
13	Instruction Test Data	Software	AF	5.7	Various		3,000,000	80
14	Memory Test	Software	AF	5.6.4	1750 ASM	500	1000	5
15	Register Test	Software	AF	5.6.5	1750 ASM	200	400	5
16	Derived Address Tests	Software	AF	5.6.6	1750 ASM	1500	1000	45
17	I/O and Interrupt Test	Software	AF	5.6.7	1750 ASM	1500	3000	45
18	Jump and Branch Tests	Software	AF	5.6.8	1750 ASM	1000	1500	2
19	Random Sequencer Test	Software	AF	5.6.9	Program Generated		TBD	20
20	Context Test(s)	Software		5.8.10	1750 ASM, 173, Other			TBD
21	Test and Simulator Validation	Software	AF	5.8.				50
22	AFAL (DAIS) ATP	Software	AF	5.8.3	1750 ASM			10

FIGURE 5.9. TEST RESOURCES TO BE DEVELOPED FOR ISA CERTIFICATION

## 6. REFERENCES

Advanced SMITE Training Manual (1979), Contract F30602-78-C-0016, CDRL 007 and CDRL 010, September.

Barbacci, M. R., Dietz, W. B., and Szewerenko, L. J. (1979) "Specification, Evaluation, and Validation of Computer Architectures Using Instruction Set Processor Descriptions", Proc. 4th International Symposium Computer Hardware Description Languages, October, pages 14 - 20.

Barbacci, M. R. and Parker, R. A. (1978) "Using Emulation to Verify Formal Machine Descriptions", COMPUTER, Volume II, No. 5, May.

Budd, T. A., DeMillo, R., Lipton, R. J., and Sayward, F. (1978) "The Design of a Prototype Mutation System for Program Testing", Proc. National Computer Conference.

Clark, N. B. and Troutman, M. A. (1979) "The System Architecture Evaluation Facility, An Emulation Facility at Rome Air Development Center", preprint.

Clary, J. B. and Smith, F. M. (1979) "Verification of Built-In-Test Performance in Modular Digital Systems using Instruction Set Processor (ISP) Language Descriptions", Proc. AUTOTESTCON '79, September, pages 96 - 101.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978) "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, April, pages 34 - 41.

Goodenough, J. B. and Gerhart, S. L. (1975) "Toward a Theory of Test Data Selection", Proc International Conference on Reliable Software, April, pages 493 - 510.

Howden, W. E. (1975) "Methodology for the Generation of Program Test Data", IEEE Trans on Computers, Volume C-24, No. 5, May, pages 554 - 559.

Ramamoorthy, C. V. and Ho, S. F. (1976) "On the Automated Generation of Program Test Data", IEEE Trans on Software Engineering, Volume SE-2, No. 4, December, pages 293 - 300.

SA 301 310 (1979) "AN/AYK-15A USER CONSOLE Interface Control Document", October.

SA 421 206 (1979) "Computer Program Development Specification for the AN/AYK-15A Acceptance Test Program", June.

SA 701 311 (1980) "PMIU Development Specification", Type (B5), Preliminary, January.

Shiva, S. G. (1979) "Computer Hardware Description Languages --A Tutorial", Proc. IEEE, Volume 67, No. 12, December, pages 1605 - 1615.

Sperry Univac (1979) "Digital Processor AN/AYK-15A Acceptance Test Procedures For", Specification No. 7314458, Preliminary, November 5.

Sperry Univac (1979) "AN/AYK-15A Acceptance Test Program", Computer Product Specification, Data Item Description No. DI-E-3120A, CDRL 44, Contract F33615-79-C-1910.

Tasar, O. and Tasar, V. (1977) "A Study of Intermittent Faults in Digital Computers", Proc. National Computer Conference, pages 807 - 811.

## APPENDIX A

### NON-OPTIONAL ASPECTS OF MIL-STD-1750 WHICH ARE NOT WELL ENOUGH DEFINED TO BE TESTED

#### 1. From Paragraph 4.1.7 Results on Fixed Point Overflow

Although ADD and SUBTRACT may reasonably be assumed from the example to wrap around in the conventional twos complement fashion, there is no such obvious result from MULTIPLY and DIVIDE Overflow, which must be assumed to produce an undefined result.

#### 2. From Paragraph 4.3.2 Special Registers

Mention is made once here of the "input/output register". Since it is never mentioned again, it remains undefined.

#### 3. From Paragraph 4.3.2.3 Fault Register

Since the PIO channel is not clearly defined (see 4.B) below), bit 3, PIO channel parity error is of some concern, although the function of a parity error is clear enough.

#### 4. From Paragraph 5 DETAIL REQUIREMENTS

- a) For all Double Precision, Floating Point, and Extended Floating Point instructions in Register Direct Addressing Mode: It is not clear whether or not it is possible and/or required for the operands to be the same or to overlap. The instructions under consideration are:

Double Precision (where RB = RA-1, RA, or RA+1)

DAR	DMR	DABS	DCR
DSR	DDR	DNEG	DLR

Floating Point (where RB = RA-1, RA, or RA+1)

FAR      FMR      FABS      RCR

FSR      FDR      FNEG      FIX\*

\* (where RB = RA-1 or RA only)

Extended Floating Point (where RB = RA-2, RA-1, RA, RA+1,  
RA+2)

EFAR    EFMR    EFCR    EFLT\*

EFSR    EFDR                    EFIX\*\*

\* (where RB = RA-1, RA, RA+1, or RA+2 only)

\*\* (where RB = RA-2, RA-1, RA, or RA+1 only)

- b) For the IN and OUT instructions related to the PIO channel (IN RA,PI,RX, and OUT RA,PO,RX), the legal range of values of RX (i.e., the number of ports) is not defined.
- c) For the OUT instruction: reset normal power up discrete bit (OUT RA,RNS), the meaning, location, and ability to read this bit are all undefined.
- d) For the breakpoint instruction (BPT), there is mention, but no definition of "the maintenance console".
- e) For all instructions of address mode DX, can RA = RX? For example, with the jump to subroutine command:
- JS    RA,LABEL,RX

The register transfer description is (p 114):

(RA) ← (IC) + 2

(IC) ← DA

where DA = LABEL + (RX). If RA = RX, then the result of the jump depends upon whether the calculation of DA is done before or after the calculation of (RA).

- f) For IN and OUT instructions regarding Timers A and B, no mention is made of when (or even whether) an interrupt is to be generated. In fact, the only mention of timer interrupts is in Table I. Interrupt definitions.
- g) For all instructions, it would be useful to clarify if there is an implied (though not stated) specification that "all registers and storage locations not expressly affected according to statements in 5. DETAIL REQUIREMENTS shall be unchanged by the operation of the instruction".

APPENDIX B

OPTIONAL ASPECTS OF MIL-STD-1750

1. From Paragraph 4.3.2.3 Fault Register:

Bits set by optional equipment.

<u>Bit</u>	
0	CPU Memory Protect
1	DMA Memory Protect
7	Other I/O Errors
13	Built In Test Equipment (BITE) Error
14-15	BITE Optional Bits

2. From Paragraph 4.4.3 Memory Parity

3. From Paragraph 4.4.4 Memory Block Protect

4. From Paragraph 4.6.3 Optional Input/Output Commands:  
see below.

5. From Paragraph 5. DETAIL REQUIREMENTS

Optional Input Commands

TPIO (I/O Buffer)	RDOR (Discrete Output Register)
RDI (Discrete Input)	CI (Console Interface Word)
RMP (Memory Protect RAM)	RCS (Console Interface Status)

Optional Output Commands

CO (Console Interface Word)	DMAE (Enable DMA)
CLC (Clear Console Interface Word)	DMAD (Disable DMA)
GO (Trigger GO)	DSUR (Disable Start Up ROM)
MPEN (Memory Protect Enable)	OD (Discrete Output)
ESUR (Enable Start Up Rom)	LMP (Load Memory Protect)



## APPENDIX C

### MIL-STD-1750 SIMULATOR SUBROUTINES

#### MASTER CLEAR

Call ISAMC

#### INITIATE EXECUTION

Call ISAXQT (NCYCLS)

NCYCLS - The number of instructions to be executed. If NCYCLS is less than one, execution will continue until a breakpoint is encountered.

#### TRANSFER DATA TO SIMULATED MEMORY

Call ISAMW (IBUF, NBUF, IADDR)

IBUF - Array of data to be transferred.

NBUF - Number of words to transfer.

IADDR - Address of data in simulated memory.

#### TRANSFER DATA FROM SIMULATED MEMORY

Call ISAMR (IBUF, NBUF, IADDR)

IBUF - Array to receive data.

NBUF - Number of words to be transferred.

IADDR - Address of data in simulated memory.

#### TRANSFER DATA TO SIMULATED REGISTERS

Call ISARW (IBUF, NBUF, IREG)

IBUF - Array of data to be transferred.

NBUF - Number of words to transfer.

IREG - Register number of first register (See Page C-4).

#### TRANSFER DATA FROM SIMULATED REGISTER

Call ISARR (IBUF, NBUF, IREG)

IBUF - Array to receive data.

NBUF - Number of words to transfer.

IREG - Register number of first register (See Page C-4).

#### SET THE SPECIFIED BREAKPOINT

Call ISAWB (NBKPT, ITYPE, ILOA, IHIA, ILOV, IHIV, MASK)

NBKPT - Breakpoint number between one and ten.

ITYPE - Type of breakpoint:

- Ø - ignore entry
- 1 - instruction
- 2 - derived operand
- 3 - instruction or operand
- 4 - register
- 5 - transfer of control
- 6 - write reference
- 7 - read reference

ILOA - Lower address (register) limit

IHIA - Upper address (register) limit

ILOV - Lower value limit

IHIV - Upper value limit

MASK - Mask to AND with value before testing limits

#### RETURN INFORMATION ABOUT THE BREAKPOINT THAT TERMINATED EXECUTION

Call ISARB (NBKPT, ITYPE, ILOA, IHIA, ILOV, IHIV, MASK, IADDR,  
IVAL)

NBKPT - The breakpoint number that halted the simulation.

A value of Ø indicates that a breakpoint instruction was encountered.

ITYPE - The type of breakpoint.

ILOA - Lower address (register) limit.

IHIA - Upper address (register) limit.

ILOV - Lower value limit.

IHIV - Upper value limit

MASK - Mask applied to value.

IADDR - Address or register number where breakpoint occurred.

IVAL - Actual data value encountered.

MIL-STD-1750 SIMULATOR REGISTER NUMBERS

<u>REGISTER NUMBER</u>	<u>REGISTER DESCRIPTION</u>
0	General Register R0
1	General Register R1
2	General Register R2
3	General Register R3
4	General Register R4
5	General Register R5
6	General Register R6
7	General Register R7
8	General Register R8
9	General Register R9
10	General Register R10
11	General Register R11
12	General Register R12
13	General Register R13
14	General Register R14
15	General Register R15
16	Instruction Counter (IC)
17	Status Word (SW)
18	Fault (FT)
19	Interrupt Mask (IM)
20	Pending Interrupt (PI)
21	Interrupt Enable
22	Power Up Discrete
23	Timer A
24	Timer B
25*	Output Buffer Register (IOR)
26*	Console Output Register
27*	Console Status Register

<u>REGISTER NUMBER</u>	<u>REGISTER DESCRIPTION</u>
28*	Go Indicator
29*	Memory Protect Enable
30*	Start Up ROM Enable
31*	DMA Enable
1000 + XXX*	Memory Protect RAM Number XXX
2000 + XXX*	Discrete Input Register Number XXX
3000 + XXX*	Discrete Output Register Number XXX
4000 + XXX*	Programmed I/O Register Number XXX

\* Implementation of these registers is optional

APPENDIX D

PROTOCOL HANDLER SUBROUTINES

INITIALIZE HANDLERS

Call UUTMC

INITIATE UUT EXECUTION

Call UUTXQT

TERMINATE UUT EXECUTION

Call UUTSTP

TRANSFER DATA TO UUT MEMORY

Call UUTMW (IBUF, NBUF, IADDR)

IBUF - Array of data to be transferred.

NBUF - Number of words to transfer.

IADDR - Address of data in UUT memory.

TRANSFER DATA FROM UUT MEMORY

Call UUTMR (IBUF, NBUF, IADDR)

IBUF - Array to receive data.

NBUF - Number of words to be transferred.

IADDR - Address of data in UUT memory.

TRANSFER DATA TO UUT REGISTERS

Call UUTRW (IBUF, NBUF, IREG)

IBUF - Array of data to be transferred.

NBUF - Number of words to transfer.

IREG - Register of number of first register (See Appendix D).

TRANSFER DATA FROM UUT REGISTERS

Call UUTRR (IBUF, NBUF, IREG)

IBUF - Array to receive data.

NBUF - Nuber of words to transfer.

IREG - Register number of first register (See Appendix D).

#### SELECT UUT DATA LINK

Call UUTSW (PATH)

PATH - Data Link to UUT.

Ø - MIL-STD-1750 Simulator.

1 - MIL-STD-1553B.

2 - RS-232C.

#### SET A UUT INTERRUPT

Call UUTINT (PIW)

PIW - Pending interrupt word.

#### SINGLE STEP UUT

Call UUTSS (NWRDS, INST)

NWRDS - The number of UUT words in the instruction (one or two).

INST - An array containing the MIL-STD-1750 instruction to be executed.

#### CHECK UUT STATUS

Call UUTCS (CODE)

CODE - Coded status word

< Ø - No status has been received.

Ø - The UUT is idle, no status can be received.

>Ø - The number of words in the status message.


#### READ UUT STATUS

Call UUTRS (IBUF, NBUF)

IBUF - Array to receive status message.

NBUF - Number of words to be transferred.

DOCUMENT NO.  
PX 13243

SPERRY  UNIVAC

## APPENDIX E

### SAMPLE MACRO ASSEMBLY CODE FOR GENERATION OF TEST PROGRAMS AND DATA

PAGE

E-1



The following pages of MIL-STD-1750 assembly code illustrate the use of macro capabilities for generation of test data and repetitive code segments. Many pages of output code have been removed to conserve paper in repetitive areas. The following index can be used to locate specific items of interest.

<u>PAGE</u>	<u>CONTENT</u>
	Initialization of assembler to MIL-STD-1750 generation
1	Macro to perform setup of data tables (see page 120 also)
2	Macros to generate a simple test of all combinations of RA, (RA), and DO for a specified OP code and list of values for (RA) and DO.
3	The macro reference that specifies the instruction to be tested (ADD), the number of values of (RA) and DO, and the table addresses for (RA), DO, and known results
61	The macro reference equivalent to the one on page 3 for the subtract instruction
119	A dummy error routine called if an error is detected.
120 - 122	Macro definitions to generate tables of operands and expected results for ADD, Subtract, AND, OR, and XOR operations.
123	Macro reference to generate operand tables and compute results using the macros defined on pages 120 - 122.



```

11. /
12. P
13. OPSS*
14.
15. . THIS MACRO PERFORMS THE SETUP OF THE OPERAND TABLES
16. .
17. . CODING FORMAT:
18. . OPSS #1,#2
19. .
20. . INPUTS:
21. . #1 ::= FIRST OPERAND
22. . #2 ::= SECOND OPERAND
23. .
24. . OUTPUTS:
25. . OPERANDS ARE PLACED IN SUBSCRIPTED SYMBOLS OP1 & OP2
26. . THE INDEX INTO OP1 & OP2 (NP) IS INCREMENTED ONCE FOR
27. . EACH REFERENCE TO THIS MACRO
28. .
29. . OP1, 'NP' EQU P(1,1)
30. . OP2*(NP) EQU P(1,2)
31. . DO PASS:(-0)=PASS:(1) ,NP*
    .
    . SET$ NP+1
    .
    . END

```

```

32. /
33. P
34. .
35. . INPUTS
36. .
37. . P(1,1) ::= INSTRUCTION OPCODE
38. . P(1,2) ::= SIZE OF OPERAND VECTORS
39. . P(1,3) ::= ADDRESS OF OP1 VECTOR
40. . P(1,4) ::= ADDRESS OF OP2 VECTOR
41. . P(1,5) ::= ADDRESS OF RESULT MATRIX
42. .
43. . OP$*
44. . J DO P(1,2)+1, 1 DO P(1,2)+1, REG P(1,1).P(1,3)+J-1,P(1,4)+I-1,;
45. . +(J+P(1,2))+P(1,5)+I-1-P(1,2)
46. . END
47. . P
48. . MACRO
49. .
50. . INPUTS:
51. .
52. . P(1,1) ::= INSTRUCTION OPCODE
53. . P(1,2) ::= ADDRESS OF OP1 VECTOR
54. . P(1,3) ::= ADDRESS OF OP2 VECTOR
55. . P(1,4) ::= ADDRESS OF RESULT MATRIX
56. .
57. . REG*
58. .
59. . NAME
60. . TXT
61. . X,132,132
62. . EDIT
63. . A,45,'ALL 16 REGISTERS FOR THE OPERAND PAIR AT'
64. . EDIT
65. . O,60,P(1,2)
66. . EDIT
67. . O,80,P(1,3)
68. . EDIT
69. . A,65,' AND '
70. . PRT
71. . 2
72. . OUTPUT ON PASS TWO
73. . TXT
74. .
75. . DO 16, INSTR$ P(1,1),I-1,P(1,2),P(1,3),P(1,4)
76. . END
77. . P
78. . MACRO
79. .
80. . INPUTS:
81. .
82. . P(1,1) ::= INSTRUCTION OPCODE
83. . P(1,2) ::= REGISTER TO USE
84. . P(1,3) ::= ADDRESS OF OP1 VECTOR
85. . P(1,4) ::= ADDRESS OF OP2 VECTOR
86. . P(1,5) ::= ADDRESS OF RESULT MATRIX
87. .
88. . INSTR$*
89. . NAME
90. . L
91. . P(1,2),P(1,3)
92. . FORM
93. . B,4,4,16
94. . XXX
95. . P(1,1),P(1,2),O,P(1,4) . PERFORM INSTRUCTION
96. . C
97. . P(1,2),P(1,5)
98. . JC
99. . 5.ERROR+P(1,2)
100. . END

```

## DATA GENERATION MACROS FOR MIL-STD-1750 VERIFICATION

PAGE 3

DATE 021880

. ADD OP CODE

X'AO'

000000014426

ADD.4,OP1\$TAB,OP2\$TAB,ADD\$TAB

00A0

/ADD

EQU

000000014421 AND

OP\$

81.

ALL 16 REGISTERS FOR THE OPERAND PAIR AT

00

82.

000000 80 0 0  
000001 1911  
000002 A0 0 0  
000003 1916  
000004 F0 0 0  
000005 1918  
000006 70 5 0  
000007 1900  
000008 80 1 0  
000009 1911  
00000A A0 1 0  
00000B 1916  
00000C F0 1 0  
00000D 1918  
00000E 70 5 0  
00000F 1901  
000010 80 2 0  
000011 1911  
000012 A0 2 0  
000013 1916  
000014 F0 2 0  
000015 1918  
000016 70 5 0  
000017 1902  
000018 80 3 0  
000019 1911  
00001A A0 3 0  
00001B 1916  
00001C F0 3 0  
00001D 1918  
00001E 70 5 0  
00001F 1903  
000020 80 4 0  
000021 1911  
000022 A0 4 0  
000023 1916  
000024 F0 4 0  
000025 1918  
000026 70 5 0  
000027 1904  
000028 80 5 0  
000029 1911  
00002A A0 5 0  
00002B 1916  
00002C F0 5 0  
00002D 1918  
00002E 70 5 0  
00002F 1905  
000030 80 6 0  
000031 1911  
000032 A0 6 0  
000033 1916

000034 F0 6 0  
000035 1918  
000036 70 5 0  
000037 1906  
000038 80 7 0  
000039 1911  
00003A A0 7 0  
00003B 1916  
00003C F0 7 0  
00003D 1918  
00003E 70 5 0  
00003F 1907  
000040 80 8 0  
000041 1911  
000042 A0 8 0  
000043 1916  
000044 F0 8 0  
000045 1918  
000046 70 5 0  
000047 1908  
000048 80 9 0  
000049 1911  
00004A A0 9 0  
00004B 1916  
00004C F0 9 0  
00004D 1918  
00004E 70 5 0  
00004F 1909  
000050 80 A 0  
000051 1911  
000052 A0 A 0  
000053 1916  
000054 F0 A 0  
000055 1918  
000056 70 5 0  
000057 190A  
000058 80 B 0  
000059 1911  
00005A A0 B 0  
00005B 1916  
00005C F0 B 0  
00005D 1918  
00005E 70 5 0  
00005F 1903  
000060 80 C 0  
000061 1911  
000062 A0 C 0  
000063 1916  
000064 F0 C 0  
000065 1918  
000066 70 5 0  
000067 190C  
000068 80 D 0  
000069 1911  
00006A A0 D 0  
00006B 1916  
00006C F0 D 0

00006D 1918  
00006E 70 5 0  
00006F 1900  
000070 80 E 0  
000071 1911  
000072 A0 E 0  
000073 1916  
000074 F0 E 0  
000075 1918  
000076 70 5 0  
000077 190E  
000078 80 F 0  
000079 1911  
00007A A0 F 0  
00007B 1916  
00007C F0 F 0  
00007D 1918  
00007E 70 5 0  
00007F 190F

ALL 16 REGISTERS FOR THE OPERAND PAIR AT 0000000014421 AND 0000000014427

000080 80 0 0  
000081 1911  
000082 A0 0 0  
000083 1917  
000084 F0 0 0  
000085 191C  
000086 70 5 0  
000087 1900  
000088 80 1 0  
000089 1911  
00008A A0 1 0  
00008B 1917  
00008C F0 1 0  
00008D 191C  
00008E 70 5 0  
00008F 1901  
000090 80 2 0  
000091 1911  
000092 A0 2 0  
000093 1917  
000094 F0 2 0  
000095 191C  
000096 70 5 0  
000097 1902  
000098 80 3 0  
000099 1911  
00009A A0 3 0  
00009B 1917  
00009C F0 3 0  
00009D 191C  
00009E 70 5 0  
00009F 1903  
0000A0 80 4 0  
0000A1 1911  
0000A2 A0 4 0

83. 0080

/SUB EQU

X'90'

ALL 16 REGISTERS FOR THE OPERAND PAIR AT 000000014421 AND 000000014426

84.

SUB.4,OP1\$TAB,OP2\$TAB,SUB\$TAB

000C80 R0 0 0  
000C81 1911  
000C82 B0 0 0  
000C83 1916  
000C84 F0 0 0  
000C85 1934  
000C86 70 5 0  
000C87 1900  
000C88 R0 1 0  
000C89 1911  
000C8A B0 1 0  
000C8B 1916  
000C8C F0 1 0  
000C8D 1934  
000C8E 70 5 0  
000C8F 1901  
000C90 R0 2 0  
000C91 1911  
000C92 B0 2 0  
000C93 1916  
000C94 F0 2 0  
000C95 1934  
000C96 70 5 0  
000C97 1902  
000C98 R0 3 0  
000C99 1911  
000C9A B0 3 0  
000C9B 1916  
000C9C F0 3 0  
000C9D 1934  
000C9E 70 5 0  
000C9F 1903  
000CA0 R0 4 0  
000CA1 1911  
000CA2 B0 4 0  
000CA3 1916  
000CA4 F0 4 0  
000CA5 1934  
000CA6 70 5 0  
000CA7 1904  
000CA8 R0 5 0  
000CA9 1911  
000CAA B0 5 0  
000CAB 1916  
000CAC F0 5 0  
000CAD 1934  
000CAE 70 5 0  
000CAF 1905  
000CB0 R0 6 0  
000CB1 1911  
000CB2 B0 6 0  
000CB3 1916



0018E4 F0 C 0  
0018E5 1948  
0018E6 70 5 0  
0018E7 190C  
0018E8 80 D 0  
0018E9 1915  
0018EA 80 D 0  
0018EB 191A  
0018EC F0 D 0  
0018ED 1948  
0018EE 70 5 0  
0018EF 190D  
0018F0 80 E 0  
0018F1 1915  
0018F2 80 E 0  
0018F3 191A  
0018F4 F0 E 0  
0018F5 1948  
0018F6 70 5 0  
0018F7 190E  
0018F8 80 F 0  
0018F9 1915  
0018FA 80 F 0  
0018FB 191A  
0018FC F0 F 0  
0018FD 1948  
0018FE 70 5 0  
0018FF 190F

DATA GENERATION MACROS FOR MIL-STD-1750 VERIFICATION

DATE 021880

PAGE 119

85.  
86.  
87.

EQU \$  
DO 16 , 9PT

/  
ERROR

	1900	
001900	FF F F	
001901	FF F F	
001902	FF F F	
001903	FF F F	
001904	FF F F	
001905	FF F F	
001906	FF F F	
001907	FF F F	
001908	FF F F	
001909	FF F F	
00190A	FF F F	
00190B	FF F F	
00190C	FF F F	
00190D	FF F F	
00190E	FF F F	
00190F	FF F F	

```

88. / P MACRO
89. FINDATA* NAME
90.
91. . THIS MACRO TERMINATES THE OPERAND DEFINITION SEQUENCE AND
92. . BUILDS THE OPERAND AND RESULT TABLES.
93. .
94. . CODING FORMAT:
95. . FINDATA
96. .
97. .
98. . INPUTS:
99. . THERE ARE NO PARAMETER INPUTS TO THIS MACRO
100. . ADDITIONAL INPUTS ARE OBTAINED FROM SYMBOL NP
101. . AND SUBSCRIPTED SYMBOLS OP1 & OP2.
102. .
103. . OUTPUTS:
104. . OP1$TAB ::= OPERAND ONE TABLE
105. . OP2$TAB ::= OPERAND TWO TABLE
106. . ADD$TAB ::= ADDITION RESULT TABLE
107. . SUB$TAB ::= SUBTRACTION RESULT TABLE
108. . AND$TAB ::= LOGICAL AND RESULT TABLE
109. . OR$TAB ::= LOGICAL OR RESULT TABLE
110. . XOR$TAB ::= LOGICAL EXCLUSIVE OR RESULT TABLE
111. .
112. . NOTE ***: THE RESULT TABLES ARE BUILT BY VARYING OP2 FASTEST
113. .
114. .
115. . NP ARMS* +
116. . TXT ' ***** NUMBER OF PARAMETERS *****'
117. . NP
118. . OP1$TAB*
119. . X
120. . DO
121. . OP2$TAB*
122. . X
123. . DO
124. . TXT
125. . TXT
126. . TXT
127. . ADD$TAB*
128. . X
129. . Y
130. . ADD$LOOP* NAME
131. .
132. . X
133. . SET$
134. . DO X<NP , GO ADD$LOOP
135. . SET$
136. . DO Y=NP , GO ADD$LOOP1
137. . TXT
138. . EDIT
139. . EDIT
140. . EDIT
141. . PRT
142. . TXT
143. . GO

```

\*\*\*\*\* OPERAND ONE TABLE \*\*\*\*\*  
 \*\*\*\*\* OPERAND TWO TABLE \*\*\*\*\*  
 \*\*\*\*\* ADDITION RESULT TABLE \*\*\*\*\*  
 \*\*\*\*\* SUBTRACTION RESULT TABLE \*\*\*\*\*  
 \*\*\*\*\* LOGICAL AND RESULT TABLE \*\*\*\*\*  
 \*\*\*\*\* LOGICAL OR RESULT TABLE \*\*\*\*\*  
 \*\*\*\*\* LOGICAL EXCLUSIVE OR RESULT TABLE \*\*\*\*\*  
 \*\*\*\*\* COLUMN # 0'

0  
 0  
 OP1(X)+OP2(Y)  
 X+1  
 GO ADD\$LOOP  
 0  
 Y+1  
 GO ADD\$LOOP1  
 X,132,132  
 A,14, \*\*\*\*\* COLUMN # '  
 D,24,Y  
 2  
 ADD\$LOOP



```

201.      DO X<NP , GO OR$LOOP .
202.      SET$ 0
203.      SET$ Y+1
204.      DO Y=NP , GO OR$LOOP1 .
205.      TXT , ,
206.      EDIT X,132,132
207.      EDIT A,14,'**** COLUMN # '
208.      EDIT D,24,Y
209.      PRT 2
210.      TXT , ,
211.      GO OR$LOOP
212.
213.      OR$LOOP1* NAME
214.      TXT ' ****LOGICAL EXCLUSIVE OR RESULT TABLE ****'
215.
216.      XOR$TAB*
217.      TXT , ,
218.      TXT '**** COLUMN # 0'
219.      SET$ 0
220.      SET$ 0
221.      XOR$LOOP* NAME
222.      + OP1(X)--OP2(Y)
223.      SET$ X+1
224.      DO X<NP , GO XOR$LOOP .
225.      SET$ 0
226.      SET$ Y+1
227.      DO Y=NP , GO XOR$LOOP1 .
228.      TXT , ,
229.      EDIT X,132,132
230.      EDIT A,14,'**** COLUMN # '
231.      EDIT D,24,Y
232.      PRT 2
233.      TXT , ,
234.      GO XOR$LOOP
235.      XOR$LOOP1* NAME
      END

```

```

/ . *****
. MAIN DATA GENERATION SECTION
. *****
. ***** INITIALIZE THE OP COUNTER *****
NP      SET$    0
OP$     0,0
OP$     1,1
OP$     0'77777',0'77776'
OP$     X'8000',0'52525'
OP$     -1,-2
FINDATA
***** NUMBER OF PARAMETERS *****
249. 001910 0005
***** OPERAND ONE TABLE *****
001911 0000
001912 0001
001913 7FFF
001914 8000
001915 FFFF
***** OPERAND TWO TABLE *****
001916 0000
001917 0001
001918 7FFE
001919 5555
C0191A FFEE
***** ADDITION RESULT TABLE *****
***** COLUMN #          0
00191B 0000
00191C 0001
00191D 7FFF
00191E 8000
00191F FFFF
***** COLUMN #          1
001920 0001
001921 0002
001922 8000
001923 8001
001924 0000
***** COLUMN #          2
001925 7FFE
001926 7FFF
001927 FFED
001928 FFE
001929 7FFD
***** COLUMN #          3
00192A 5555

```

001928 5556  
00192C D554  
001920 D555  
00192E 5554

\*\*\*\* COLUMN # 4

00192F FFFE  
001930 FFFF  
001931 7FFD  
001932 7FFE  
001933 FFFD

\*\*\*\* SUBTRACTION RESULT TABLE \*\*\*\*

\*\*\*\* COLUMN # 0

001934 0000  
001935 0001  
001936 7FFF  
001937 8000  
001938 FFFF

\*\*\*\* COLUMN # 1

001939 FFFF  
00193A 0000  
00193B 7FFE  
00193C 7FFF  
00193D FFFE

\*\*\*\* COLUMN # 2

00193E 8002  
00193F 8003  
001940 0001  
001941 0002  
001942 8001

\*\*\*\* COLUMN # 3

001943 AAAB  
001944 AAAC  
001945 2AAA  
001946 2AAB  
001947 AAAA

\*\*\*\* COLUMN # 4

001948 0002  
001949 0003  
00194A 8001  
00194B 8002  
00194C 0001

\*\*\*\* LOGICAL AND RESULT TABLE \*\*\*\*

DATA GENERATION MACROS FOR MIL-STD-1750 VERIFICATION

DATE 021880 PAGE 125

\*\*\*\* COLUMN # 0

00194D 0000  
00194E 0000  
00194F 0000  
001950 0000  
001951 0000

\*\*\*\* COLUMN # 1

001952 0000  
001953 0001  
001954 0001  
001955 0000  
001956 0000

\*\*\*\* COLUMN # 2

001957 0000  
001958 0000  
001959 7FFE  
00195A 0000  
00195B 7FFE

\*\*\*\* COLUMN # 3

00195C 0000  
00195D 0001  
00195E 5555  
00195F 0000  
001960 5554

\*\*\*\* COLUMN # 4

001961 0000  
001962 0001  
001963 7FFD  
001964 8000  
001965 FFFD

\*\*\*\*\*LOGICAL OR RESULT TABLE \*\*\*\*\*

\*\*\*\* COLUMN # 0

001966 0000  
001967 0001  
001968 7FFF  
001969 8000  
00196A FFFF

\*\*\*\* COLUMN # 1

00196B 0001  
00196C 0001  
00196D 7FFF  
00196E 8001  
00196F 0000



DATA GENERATION MACROS FOR MIL-STD-1750 VERIFICATION

\*\*\*\* COLUMN # 2

001970 7FFE  
001971 7FFF  
001972 7FFF  
001973 FFFE  
001974 FFFF

\*\*\*\* COLUMN # 3

001975 5555  
001976 5555  
001977 7FFF  
001978 0555  
001979 0000

\*\*\*\* COLUMN # 4

00197A FFFE  
00197B FFFE  
00197C 0000  
00197D FFFE  
00197E 0000

\*\*\*\*LOGICAL EXCLUSIVE OR RESULT TABLE \*\*\*\*

\*\*\*\* COLUMN # 0

00197F 0000  
001980 0001  
001981 7FFF  
001982 8000  
001983 FFFF

\*\*\*\* COLUMN # 1

001984 0001  
001985 0000  
001986 7FFE  
001987 8001  
001988 0000

\*\*\*\* COLUMN # 2

001989 7FFE  
00198A 7FFF  
00198B 0001  
00198C FFFE  
00198D 8001

\*\*\*\* COLUMN # 3

00198E 5555  
00198F 5554  
001990 2AAA  
001991 0555  
001992 AAAC

•••• COLUMN # 4

250.	001993	FFE
251.	001994	FFD
252.	001995	8003
253.	001996	7FFE
254.	001997	0003
255.		1911
256.		1916
		1918
		1934
		1940
		197F
		1966

F1	EQU	OP1\$TAB	.
F2	EQU	OP2\$TAB	.
F3	EQU	ADD\$TAB	.
F4	EQU	SUB\$TAB	.
F5	EQU	AND\$TAB	.
F6	EQU	XOR\$TAB	.
F7	EQU	OR\$TAB	.
	END		

## APPENDIX F

### REVIEW OF THE AFAL (DAIS) AN/AYK-15A ATP PROGRAM DESIGN

The test methodology represented by this ATP (PA 401 205) is a controlled procedure for executing one or more times particular test modules (programs), which verify the MIL-STD-1750 functional characteristics by running instruction sequences which exercise desired aspects of the computer repertoire. Some test modules such as the memory test and power on/off sequencing require separate load modules, but most are sub-programs under the control of an executive.

The executive allows the user to specify which test modules to run, whether they should be run in sequence or iteratively, and what to do when an error is detected. The various test modules generally set a unique flag for every different error, and those error flags are examined by an error processing routine at the conclusion of each test module. Similarly, interrupt handlers set flags associated with the potential interrupts so that each test module can monitor expected and unexpected interrupts.

According to the specification (SA 421 206), there are twenty-seven processor test modules. Some are concerned with matters outside MIL-STD-1750 or aspects of MIL-STD-1750.

Others are functionally related under an organization which might be summarized as follows:

#### Processor Test Modules

1. Instruction Set
2. Registers
3. Memory
4. Input/Output

## 5. Interrupts

### Instruction Set Tests

Four of the five instruction set tests are relevant to the certification of MIL-STD-1750: individual instruction tests, indexing test, illegal instruction test, and hand test. (The benchmark or throughput test is not directly related since it is concerned with timing, a hardware consideration, and not with functional operation.) The individual instruction tests are organized with one subroutine per opcode, with the exception of the I/O commands, which are verified separately. These tests constitute most of the entire ATP, and their internal structure will be described in detail later.

The indexing test was specified to verify separately from the individual tests, that for at least one opcode which allows indexing, all possible index values would be checked for generating a legal address. The illegal instruction test verifies that illegal opcodes correctly generate a machine error interrupt and set the illegal opcode bit in the fault register. Although this test is grouped with the instruction set tests, it overlaps in its scope with the interrupt and register tests. The hand test specifies that random but legal sequences of instructions should be executed in order to check for errors which relate particularly to context effects. Presumably jump and branch instructions are excluded from this test.

The individual instruction tests are implemented and are organized into two parts which are common to all tests and three parts which are used whenever they apply to the particular opcode. The common parts are first to verify that known outputs derive from the operation of the opcode with known inputs and initial conditions, and second that no overflow or underflow

interrupts occurred when they were not supposed to. The ATP specifications about what inputs values to use are fairly brief: they say that arithmetic instructions should use selected values in each of the four quadrants and test add/subtract and multiply/divide as pairs to confirm that these operations are symmetric. In addition, bit manipulation, logic function, and control instructions are to be tested using all four bit sections of 16 bit operands and all bit changes (i.e.,  $1 \rightarrow 0$ ,  $0 \rightarrow 1$ ,  $1 \rightarrow 1$ ,  $0 \rightarrow 0$ ).

The other three parts of the individual instruction tests, which are done where applicable are 1) to verify the correct setting of all required status bits, 2) to verify the correct generation of overflow or underflow interrupts, and 3) to verify the operation of indexable instructions with a selection of different index registers and index values. Of these three tests, only the check of the condition status is a complete or exhaustive tests in the sense that all possible status word settings are confirmed. The other tests are sample checks where typical cases are verified.

In summary, the instruction set tests are organized by opcode with separate tests to verify illegal instruction operation, the indexing mode of addressing, and the execution of instructions in a random context (the hang test). The individual instruction tests are organized into parts that verify basic operation of the instruction, lack of extraneous overflow/underflow interrupts, and where relevant, correct setting of all required status bits, correct generation of overflow/underflow interrupts, and correct operation of indexing for a few test cases.

Several comments may be made about what is not specified in the instruction set tests. One thing is any direction about

the order of testing opcodes (apparently the default sequence is by opcode value) and what instructions can be used in writing tests of other instructions (in the actual code a fixed subset of unverified instructions are used to test all others). Another thing is that addressing modes other than those using an index register are not specifically tested by separate modules. Thus, for example, indirect addressing as such is not specifically tested. It is used in limited variety only, with each opcode that has indirect addressing. A third consideration is that there is no central organization of opcodes by function, such as the use of common data tables for arithmetic operations. Thus it is hard to tell whether add/subtract have been tested for symmetry and hard to examine the operands used in arithmetic or logical instructions without going to the code for each separate opcode test. Finally, the only check on whether an instruction has extraneous unwanted effects on the machine is the check on whether overflow/underflow interrupts occurred when they were not supposed to.

### Register Tests

The register test module is specified to be a program which verifies the capability to address, set and reset all possible values of the general registers and the writable special registers, namely: the status word, the instruction counter, and the interrupt mask. (The pending interrupt register is also writable, but is tested as part of the interrupt test). For each general register, the method of verification is to load from memory all possible 16 bit values, checking each time that the value loaded is correct and that no other general registers are affected. The same check is performed on the status word and the interrupt mask except that no other registers are checked for changes. For the instruction center (IC), the test is to fill core (one half at a time) with a one

instruction which increments a general register by one. Assuming the increment instruction words and was put into memory correctly, then at the end of the test the general register and the IC should have the same value, indicating that the IC incremented correctly through memory.

All of these test overlap somewhat with the intrinsic testing of registers done in all other tests where registers are referenced. Here, of course, the intent is to more thoroughly verify register operation by controlling the data values that pass through them. These tests especially highlight the fact that every test in the ATP assumes the correct operation of the instructions used to conduct the test. Since there is no ordering or hierarchical structure to the ATP test modules, the inherent redundancy of the various tests is the method employed to cross-check instructions against each other.

#### Memory Tests

Two memory tests are specified for the ATP: the first, a memory integrity test verifies the ability to address, write, and read all of memory; and the second, an illegal memory address test confirms that a machine error interrupt is generated when a memory module is removed and reference is made to an address therein.

For the memory integrity test, several bit patterns are tested (addressed, all ones, all zeros, and a worst case pattern which is of course hardware dependent). Considering the memory space (65K) divided into four portions, the bit pattern tests are conducted over portions 1 and 2, 3 and 4, 2 and 3, and 1 and 4. After tests in each portion, the rest of memory is to be checksummed to see if illegal changes to memory occurred.

### Input/Output Tests

The I/O tests focus on the INPUT and OUTPUT instruction codes for programmed I/O ports (PIO) and the two timers. The other I/O commands are verified in either the register tests or the interrupt tests. Testing the PIO capability involves a hardware connection from the output port to the input port which allows an IN PI command to verify data written via an OUT PO. Confirming all possible 16 bit patterns completes this test. Since the two timers have specified time increments, the timer test also involves external hardware to confirm those values. The specified procedure is to start the timers with known values, stop them after known intervals, and compare their contents. A second timer test apparently compares the actual and expected times when a timer interrupt occurs. The point of timer interrupt is not defined by MIL-STD-1750. A final part of the I/O tests tries all illegal commands for IN and OUT, verifying that they give a machine error interrupt.

### Interrupt Tests

The interrupt tests are intended to verify those interrupts not previously tested in the register and I/O tests. According to the standard, the only remaining interrupts are for optional aspects of 1750. However, IN and OUT commands related to the pending interrupt register and fault register, and the OUT commands enable and disable interrupts all remain to be specifically verified at this point. The ATP specification states that external hardware should be used to exercise these registers with associated IN and OUT commands with interrupts masked and unmasked. With respect to testing the various bits in these registers, this test is similar to the register tests.



## APPENDIX G

### Review of the Sperry Univac AN/AYK-15A Acceptance Test Program Design

The Confidence Test (1979) represents another test methodology which verifies functional characteristics of MIL-STD-1750. Although aimed more as a factory acceptance test than as a design certification test, it offers some alternative approaches from those of the ATP. Structurally, the Confidence Test is organized as two core loads: a package containing 24 ordered test modules defined by 1750 functional capabilities and a standalone memory test.

The first of the ordered test modules performs a basic operation test of 12 load, store, add, jump, and compare instructions, plus the reading and writing of the Status Register and the Interrupt Mask Register. Successive modules then use these partially verified instructions to perform functional tests beginning with jump and branch instructions, single precision load and store, and so forth, ending with tests of Base Register Relative Addressing, Interrupts, Timers, and the Fault Register. A complete list of the modules appears in Figure I-1. The modules are actually written so they can be called as subroutines from some executive, but because they build on each other in terms of capabilities verified, they would be run in order at least the first time.

The effect of this ordering is that test modules after the first few are free to utilize fairly sophisticated code segments to provide both a good mix of instructions tested and a variety of techniques such as loops, access of data tables, etc. with which to write efficient code. The separation of test data from test code is effective in making the code

readable. And the freedom to write well-structured routines gained from the ability to use much of the instruction set means that the resulting test code will be more likely to be correct and easier to debug.

Resulting style of coding is more similar to that used in real mission software than is the regular, simple structure of code in the ATP. For example, the ATP tested a multiply instruction by setting up the operands, doing one multiply, then checking the results. Thus the density of multiply executions during the test is very low. The Confidence Test places multiply instructions in loops and also performs them in sequence. This offers the possibility of more easily testing the instruction more often and of testing the instruction in a repetitive context which in some sense exercises the multiply operation more heavily.

Because it was written for a particular 1750 implementation, the Confidence Test is oriented towards a known computer architecture. The Univac Confidence Test took advantage of this knowledge, taking short cuts in the test design whenever common hardware was known. Thus using the functional organization of the test modules, the test of an Add could, for example, fully verify the add function in one address mode and make assumptions that the adder worked in tests of other address modes.

DOCUMENT NO.

PX-13243

SPERRY  UNIVAC

APPENDIX H

SORTED LIST OF MIL-STD-1750 INSTRUCTIONS

PAGE

H-1

The following list of MIL-STD instructions is sorted in the order:

- . addressing mode
- . data format
- . instruction class

The purpose of using this sort order is to assist in partitioning of individual instruction test loops as outlined in Section 5.6.3.

Column headings and associated data values are as follows:

- OC - Operation Code in hexadecimal. The letter B following the code indicates that the value of the active base register (Ø-3) should be added to the operation code. The third hexadecimal character on base relative and immediate instructions indicates the operation code extension value.
- MN - Instruction mnemonic from MIL-STD-1750
- AM - Addressing Mode
- B - Base relative
  - BX - Base relative indexed
  - I - Indirect and Indirect indexed
  - IR - IC relative
  - IM - Immediate long and Immediate long indexed
  - IS - Immediate short positive and Immediate short negative
  - M - Memory direct and Memory direct indexed
  - R - Register direct
  - S - Special format

## DF - Data Format

- B - Bit
- C - Condition
- D - Double precision (32 bits)
- E - Extended floating point (48 bits)
- F - Floating point
- I - Integer (16 bit)
- S - Special format

## IN - Interrupts Generated

- N - No interrupts
- O - Integer overflow
- FOU - Floating overflow and underflow
- OFU - Integer overflow, floating point underflow
- E - Instruction error

## CS - Condition Status

- N - Status word not affected
- C - Carry and status are set
- S - Status is set, carry will be 0

## IC - Instruction Class

- C - Compare class
- J - Jump class
- L - Load or arithmetic class
- R - Memory replace class
- S - Store class
- X - Special class

DOCUMENT NO.

PX 13243

SPERRY UNIVAC

OC	MIN	AM	DF	IN	CS	IC	DESCRIPTION	PAGE
----	-----	----	----	----	----	----	-------------	------

44				E			ILLEGAL OPERATION	
45				E			ILLEGAL OPERATION	
46				E			ILLEGAL OPERATION	
47				E			ILLEGAL OPERATION	
4A0				E			ILLEGAL OPERATION	
4AC				E			ILLEGAL OPERATION	
4AD				E			ILLEGAL OPERATION	
4AE				E			ILLEGAL OPERATION	
4AF				E			ILLEGAL OPERATION	
4B				E			ILLEGAL OPERATION	
4C				E			ILLEGAL OPERATION	
4D				E			ILLEGAL OPERATION	
4E				E			ILLEGAL OPERATION	
4F				E			ILLEGAL OPERATION	
5B				E			ILLEGAL OPERATION	
5D				E			ILLEGAL OPERATION	
5F				E			ILLEGAL OPERATION	
64				E			ILLEGAL OPERATION	
69				E			ILLEGAL OPERATION	
77				E			ILLEGAL OPERATION	
7C				E			ILLEGAL OPERATION	
95				E			ILLEGAL OPERATION	
AD				E			ILLEGAL OPERATION	
AE				E			ILLEGAL OPERATION	
AF				E			ILLEGAL OPERATION	
BD				E			ILLEGAL OPERATION	
BE				E			ILLEGAL OPERATION	
BF				E			ILLEGAL OPERATION	
CC				E			ILLEGAL OPERATION	
CD				E			ILLEGAL OPERATION	
CE				E			ILLEGAL OPERATION	
CF				E			ILLEGAL OPERATION	
DC				E			ILLEGAL OPERATION	
DD				E			ILLEGAL OPERATION	
DE				E			ILLEGAL OPERATION	
DF				E			ILLEGAL OPERATION	
EE				E			ILLEGAL OPERATION	
EF				E			ILLEGAL OPERATION	
F5				E			ILLEGAL OPERATION	
FC				E			ILLEGAL OPERATION	
FD				E			ILLEGAL OPERATION	
FE				E			ILLEGAL OPERATION	

04B	DLB	B	D	N	S	L	DOUBLE LOAD	92
0CB	DSTB	B	D	N	N	S	DOUBLE STORE	101
3CB	FCB	B	F	N	S	C	FLOATING POINT COMPARE	59
20B	FAB	B	F	FOU	S	L	FLOATING POINT ADD	35
2CB	FDB	B	F	FOU	S	L	FLOATING POINT DIVIDE	41
28B	FMB	B	F	FOU	S	L	FLOATING POINT MULTIPLY	39

PAGE

H-4

DOCUMENT NO.  
PX 13243

SPERRY UNIVAC

OC	MN	AM	DF	IN	CS	IC	DESCRIPTION	PAGE
24B	FSB	B	F	FOU	S	L	FLOATING POINT SUBTRACT	37
38B	CB	B	I	N	S	C	INTEGER COMPARE	57
10B	AB	B	I	O	C	L	INTEGER ADD	25
34B	ANDB	B	I	N	S	L	LOGICAL AND	67
00B	LB	B	I	N	S	L	INTEGER LOAD	91
30B	ORB	B	I	N	S	L	INCLUSIVE LOGICAL OR	66
14B	SBB	B	I	O	C	L	INTEGER SUBTRACT	27
08B	STB	B	I	N	S	S	INTEGER STORE	100
18B	MB	B	ID	O	S	L	INTEGER MULTIPLY 32 BIT PROD.	30
1CB	DB	B	IS	O	S	L	INTEGER DIVIDE 32 BITS	33
40B1	DLBX	BX	D	N	S	L	DOUBLE LOAD	92
40B3	DSTX	BX	D	N	N	S	DOUBLE STORE W BASE REG	101
40B7	DBX	BX	DS	O	S	L	INTEGER DIVIDE 32 BITS	33
40BD	FCBX	BX	F	N	S	C	FLOATING POINT COMPARE	59
40B8	FABX	BX	F	FOU	S	L	FLOATING POINT ADD W BASE REG	35
40BB	FDBX	BX	F	FOU	S	L	FLOATING POINT DIVIDE	41
40BA	FMBX	BX	F	FOU	S	L	FLOATING POINT MULTIPLY	39
40B9	FSBX	BX	F	FOU	S	L	FLOATING POINT SUBTRACT	37
40BC	CBX	BX	I	N	S	C	INTEGER COMPARE	57
40B4	ABX	BX	I	O	C	L	INTEGER ADD	25
40BE	ANDX	BX	I	N	S	L	LOGICAL AND	67
40B0	LBX	BX	I	N	S	L	INTEGER LOAD	91
40BF	ORBX	BX	I	N	S	L	INCLUSIVE LOGICAL OR	66
40B5	SBBX	BX	I	O	C	L	INTEGER SUBTRACT	27
40B2	STBX	BX	I	N	S	S	INTEGER STORE	100
40B6	MBX	BX	ID	O	S	L	INTEGER MULTIPLY 32 BIT PROD.	30
8E	LLBI	I	8	N	S	L	LOAD FROM LOWER BYTE INDIRECT	95
8D	LUBI	I	8	N	S	L	LOAD FROM UPPER BYTE INDIRECT	94
9E	SLBI	I	8	N	N	S	STORE IN LOWER BYTE INDIRECT	106
9D	SUBI	I	8	N	N	S	STORE IN UPPER BYTE INDIRECT	105
55	RBI	I	B	N	N	R	RESET BIT	85
52	SBI	I	B	N	N	R	SET BIT	84
58	TBI	I	B	N	S	R	TEST & SET BIT	36
71	JCI	I	C	N	N	J	JUMP ON CONDITION INDIRECT	113
38	DLI	I	D	N	S	L	DOUBLE LOAD INDIRECT	92

OC	MN	AM	DF	IN	CS	IC	DESCRIPTION	PAGE
98	DSTI	I	D	N	N	S	DOUBLE STORE INDIRECT	101
92	STCI	I	H	N	N	S	STORE HEX CONSTANT INDIRECT	103
84	LI	I	I	N	S	L	INTEGER LOAD INDIRECT	91
94	STI	I	I	N	N	S	INTEGER STORE INDIRECT	101
4A5	DIM	IM	DS	O	S	L	INTEGER DIVIDE 32 BITS	33
4AA	CIM	IM	I	N	S	C	INTEGER COMPARE	57
4A1	AIM	IM	I	O	C	L	INTEGER ADD	25
4A7	ANDM	IM	I	N	S	L	LOGICAL AND	67
35	LIM	IM	I	N	S	L	INTEGER LOAD IMMEDIATE	91
4A3	MIM	IM	I	O	S	L	INTEGER MULTIPLY 32 BIT PROD.	30
4A4	MSIM	IM	I	O	S	L	INTEGER MULTIPLY 16 BIT PROD.	29
4AB	NIM	IM	I	N	S	L	LOGICAL NAND	69
4A8	ORIM	IM	I	N	S	L	INCLUSIVE LOGICAL OR	66
4A2	SIM	IM	I	O	C	L	INTEGER SUBTRACT IMMEDIATE	27
4A9	XORM	IM	I	N	S	L	EXCLUSIVE LOGICAL OR	63
4A6	DVIM	IM	IS	O	S	L	INTEGER DIVIDE 16 BITS	32
75	BEZ	IR	C	N	N	J	BRANCH EQUAL TO (ZERO)	117
7B	BGE	IR	C	N	N	J	BRANCH GREATER THAN OR = 0	122
79	BGT	IR	C	N	N	J	BRANCH GREATER (ZERO)	120
78	BLE	IR	C	N	N	J	BRANCH LESS THAN OR EQUAL (0)	119
76	BLT	IR	C	N	N	J	BRANCH LESS THAN (ZERO)	118
7A	BNZ	IR	C	N	N	J	BRANCH NOT EQUAL TO (ZERO)	121
74	BR	IR	C	N	N	J	BRANCH UNCONDITIONAL	116
F3	CISN	IS	I	N	S	C	COMPARE NEG. HEX DIGIT	57
F2	CISP	IS	I	N	S	C	COMPARE HEX DIGIT	57
A2	AISP	IS	I	O	C	L	ADD A POSITIVE HEX	25
D3	DISN	IS	I	O	S	L	INTEGER HEX DIVIDE 16 BITS	32
D2	DISP	IS	I	O	S	L	INTEGER POS HEX DIVIDE 16 BIT	32
83	LISN	IS	I	N	S	L	HEX DIGIT LOAD IMMEDIATE	31
82	LISP	IS	I	N	S	L	HEX DIGIT LOAD IMMEDIATE	31
C3	MISN	IS	I	O	S	L	INTEGER NEGATIVE HEX DIGIT	30
C2	MISP	IS	I	O	S	L	MULTIPLY HEX DIGIT	29
B2	SISP	IS	I	O	C	L	SUBTRACT A HEX DIGIT	27
8C	LLB	M	8	N	S	L	LOAD FROM LOWER BYTE	95
8B	LUB	M	8	N	S	L	LOAD FROM UPPER BYTE	94
9C	STLB	M	8	N	N	S	STORE INTO LOWER BYTE	106
9B	STUB	M	8	N	N	S	STORE INTO UPPER BYTE	105
53	PB	M	B	N	N	R	RESET BIT	35
50	SB	M	B	N	N	R	SET BIT	34
56	TB	M	B	N	S	R	TEST & SET BIT	36
59	TSB	M	B	N	S	R	TEST AND SET BIT	37



DOCUMENT NO.  
PX 13243

SPERRY UNIVAC

OC	MN	AM	DF	IN	CS	IC	DESCRIPTION	PAGE
70	JC	M	C	N	N	J	JUMP ON CONDITION	113
F6	DC	M	D	N	S	C	DOUBLE COMPARE	58
A6	DA	M	D	O	C	L	DOUBLE INTEGER ADD	26
86	DL	M	D	N	S	L	DOUBLE LOAD	92
C6	DM	M	D	O	S	L	DOUBLE PRECISION MULTIPLY	31
B6	DS	M	D	O	C	L	DOUBLE INTEGER SUBTRACT	23
96	DST	M	D	N	N	S	DOUBLE STORE	101
D6	DD	M	DS	O	S	L	DOUBLE PRECISION DIVIDE	34
D4	D	M	DS	O	S	L	INTEGER DIVIDE 32 BITS	33
FA	EFC	M	E	N	S	C	EXTENDED FLOATING COMPARE	60
AA	EFA	M	E	FOU	S	L	EXTENDED FLOATING ADD	42
DA	EFD	M	E	FOU	S	L	EXTENDED FLOATING DIVIDE	42
3A	EFL	M	E	N	S	L	EXTENDED FLOATING POINT LOAD	93
CA	EFM	M	E	FOU	S	L	EXTENDED FLOATING MULTIPLY	46
BA	EFS	M	E	FOU	S	L	EXTENDED FLOATING SUBTRACT	44
9A	EFST	M	E	N	N	S	EXTENDED FLOAT STORE	102
F8	FC	M	F	N	S	C	FLOATING POINT COMPARE	59
A8	FA	M	F	FOU	S	L	FLOATING POINT ADD	35
D8	FD	M	F	FOU	S	L	FLOATING POINT DIVIDE	41
C8	FM	M	F	FOU	S	L	FLOATING POINT MULTIPLY	39
B8	FS	M	F	FOU	S	L	FLOATING POINT SUBTRACT	37
F0	C	M	I	N	S	C	INTEGER COMPARE	57
F4	CBL	M	I	N	S	C	COMPARE BETWEEN LIMITS	61
72	JS	M	I	N	N	J	JUMP TO SUBROUTINE	114
73	SOJ	M	I	N	N	J	JUMP BACK AND COUNT	115
7F	URS	M	I	N	N	J	UNSTACK & SUBROUTINE RETURN	112
A0	A	M	I	O	C	L	INTEGER ADD	25
E2	AND	M	I	N	S	L	LOGICAL AND	67
30	L	M	I	N	S	L	INTEGER LOAD	91
7D	LDST	M	I	N	O	L	LOAD STATUS	96
C4	M	M	I	O	S	L	INTEGER MULTIPLY 32 BIT PROD.	30
C0	MS	M	I	O	S	L	INTEGER MULTIPLY 16 BIT PROD.	29
E6	N	M	I	N	S	L	LOGICAL NAND	69
E0	OR	M	I	N	S	L	INCLUSIVE LOGICAL OR	66
B0	S	M	I	O	C	L	INTEGER SUBTRACT	27
B3	DECM	M	I	O	C	R	DECREMENT MEMORY BY INTEGER	50

PAGE

H-7

OC	MN	AM	DF	IN	CS	IC	DESCRIPTION	PAGE
A3	INCM	M	I	O	C	R	INCREMENT MEMORY BY HEX DIGIT	49
90	ST	M	I	N	N	S	INTEGER STORE	100
91	STC	M	I	N	N	S	STORE POSITIVE HEX CONSTANT	103
D0	DV	M	IS	O	S	L	INTEGER DIVIDE 16 BITS	32
97	SRM	M	S	N	N	S	STORE REGISTER THROUGH MASK	104
54	RBR	R	B	N	N	L	RESET BIT	85
51	SBR	R	B	N	N	L	SET BIT, BIT IN REGISTER	84
57	TBR	R	B	N	S	L	TEST & SET BIT	86
F7	DCR	R	D	N	S	C	DOUBLE COMPARE REG.	58
A5	DABS	R	D	N	S	L	DOUBLE ABSOLUTE VALUE	52
A7	DAR	R	D	O	C	L	DOUBLE INTEGER ADD REG.	26
D7	DDR	R	D	O	S	L	DOUBLE PRECISION DIVIDE	34
37	DLR	R	D	N	S	L	DOUBLE LOAD REGISTER	92
C7	DMR	R	D	O	S	L	DOUBLE PRECISION MULTIPLY REG	31
B5	DNEG	R	D	O	S	L	DOUBLE PRECISION NEGATE REG.	55
6E	DSAR	R	D	N	S	L	DOUBLE SHIFT ARITH REG. COUNT	32
6F	DSCR	R	D	N	S	L	DOUBLE SHIFT CYCLIC REG COUNT	33
68	DSLCL	R	D	N	S	L	DOUBLE SHIFT LEFT CYCLIC	77
65	DSLCL	R	D	N	S	L	DOUBLE SHIFT LEFT LOGICAL	74
6D	DSLRL	R	D	N	S	L	DOUBLE SHIFT LOGICAL REGISTER	81
B7	DSR	R	D	O	C	L	DOUBLE INTEGER SUBTRACT	28
67	DSRA	R	D	N	S	L	DOUBLE SHIFT RIGHT ARITHMETIC	76
66	DSRL	R	D	N	S	L	DOUBLE SHIFT RIGHT LOGICAL	75
EB	EFLT	R	DE	N	S	L	32 BIT INT TO EXTENDED FLOAT	63
DS	DR	R	DS	O	S	L	INTEGER DIVIDE 32 BIT REG.	33
FB	EFCR	R	E	N	S	C	EXTENDED FLOATING COMPARE	60
AB	EFAR	R	E	FOU	S	L	EXTENDED FLOATING ADD REG	42
DB	EFDR	R	E	FOU	S	L	EXTENDED FLOATING DIVIDE	43
EA	EFIX	R	E	OFU	S	L	EXTENDED FLOAT TO FIX 32 BIT	65
CB	EFMR	R	E	FOU	S	L	EXTENDED FLT. MULTIPLY REG.	46
BB	EFSR	R	E	FOU	S	L	EXTENDED FLT SUBTRACT REG.	44
F9	FCR	R	F	N	S	C	LOATING POINT COMPARE REG	59
AC	FABS	R	F	N	S	L	FLOATING POINT ABSOLUTE VALUE	53
A9	FAR	R	F	FOU	S	L	FLOATING POINT ADD REG TO REG	35
D9	FDR	R	F	FOU	S	L	FLOATING POINT DIVIDE	41
E8	FIX	R	F	OFU	S	L	CONVERT FLOAT TO FIX 16 BITS	64
C9	FMR	R	F	FOU	S	L	FLOATING POINT MULTIPLY REG.	39
BC	FNEG	R	F	FOU	S	L	FLOATING POINT NEGATE	56
B9	FSR	R	F	FOU	S	L	FLOATING POINT SUBTRACT	37
F1	CR	R	I	N	S	C	INTEGER COMPARE REG.	57

OC	MN	AM	DF	IN	CS	IC	DESCRIPTION	PAGE
SE	TVBR	R	I	N	S	C	TEST VARIABLE BIT IN REGISTER 90	
A4	ABS	R	I	N	S	L	INTEGER ABSOLUTE VALUE	51
E3	ANDR	R	I	N	S	L	LOGICAL AND REG. TO REG.	67
A1	AR	R	I	O	C	L	INTEGER ADD REG. TO REG.	25
81	LR	R	I	N	S	L	LOAD REG. TO REG.	91
C5	MR	R	I	O	S	L	INTEGER MULTIPLY REG. TO REG.	30
C1	MSR	R	I	O	S	L	INTEGER MULTIPLY	29
B4	NEG	R	I	O	S	L	INTEGER NEGATE REGISTER	54
E7	NR	R	I	N	S	L	LOGICAL NAND REG. TO REG	69
E1	ORR	R	I	N	S	L	INCLUSIVE LOGICAL OR REG.	66
5C	RVBR	R	I	N	N	L	RESET VARIABLE BIT IN REG.	89
6B	SAR	R	I	N	S	L	SHIFT ARITH, COUNT IN REG.	79
6C	SCR	R	I	N	S	L	SHIFT CYCLIC, COUNT IN REG.	80
63	SLC	R	I	N	S	L	SHIFT LEFT CYCLIC	73
61	SLL	R	I	N	S	L	SHIFT LEFT LOGICAL	70
6A	SLR	R	I	N	S	L	SHIFT LOGICAL, COUNT IN REG.	78
B1	SR	R	I	O	C	L	INTEGER SUBTRACT REG. TO REG.	27
62	SRA	R	I	N	S	L	SHIFT RIGHT ARITHMETIC	72
61	SRL	R	I	N	S	L	SHIFT RIGHT LOGICAL	71
5A	SVBR	R	I	N	N	L	SET VARIABLE BIT IN REGISTER	89
EC	XBR	R	I	N	S	L	EXCHANGE BYTES IN REGISTER	112
E4	XOR	R	I	N	S	L	EXCLUSIVE LOGICAL OR	68
E5	XORR	R	I	N	S	L	EXCLUSIVE LOGICAL OR REG.	68
ED	XWR	R	I	N	S	L	EXCHANFW WORDS IN REGISTERS	99
E9	FLT	R	IF	N	S	L	CONVERT INTEGER TO FLT POINT	62
D1	DVR	R	IS	OP	S	L	INTEGER DIVIDE 16 BITS	32
7E	SJS	S	I	N	N	J	STACK IC, JUMP TO SUBROUTINE	111
48	IN	S	I	N	N	X	INPUT	124
49	OUT	S	I	N	N	X	OUTPUT	124
89	LM	S	M	N	N	X	LOAD MULTIPLE (0<N<15)	97
93	MOV	S	M	N	N	X	MOVE MULTIPLE WORDS	107
8F	POPM	S	M	N	N	X	POP MULTIPLE REG OFF STACK	110
9F	PSHM	S	M	N	N	X	PUSH MULTIPLE REG ONTO STACK	109
99	STM	S	M	N	N	X	STORE MULTIPLE REGISTERS	108
FF	BPT	S	N	N	N	X	BREAKPOINT	132
FF	NOP	S	N	N	N	X	NO OPERATION	131