AD-A 085 084

AD

ARSCD – CR – 8005

AD-E 400-421

ECONOMIC TRADE/OFF ANALYSIS: COMMON HARDWARE/SOFTWARE COMPUTER RESOURCES FOR THE ADVANCED ATTACK HELICOPTER (YAH-64) FIRE CONTROL SYSTEM

RAYMOND J. BRACHMAN, PROGRAM MANAGER
R. J. BRACHMAN ASSOCIATES, INC.
GEORGE CLINEFF, PROJECT ENGINEER
US ARMY ARRADCOM

US ARMY ARMAMENT RESEARCH AND DEVELOPMENT COMMAND
FIRE CONTROL AND SMALL CALIBER
WEAPON SYSTEMS LABORATORY
DOVER, NEW JERSEY

PREPARED UNDER CONTRACT
DAAK 10-79-C-0329

APRIL 1980

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ECONOMIC/TRADE-OFF ANALYSIS: COMMON HARDWARE-SOFTWARE COMPUTER RESOURCES FOR THE ADVANCED ATTACK HELICOPTER FIRE CONTROL SYSTEM (U)		5. TYPE OF REPORT & PERIOD COVERED FINAL October 1979-April 1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) R. J. BRACHMAN ASSOCIATES, INC. STAFF		8. CONTRACT OR GRANT NUMBER(s) DAAK10-79-C-0329
9. PERFORMING ORGANIZATION NAME AND ADDRESS R. J. BRACHMAN ASSOCIATES, INC. P. O. BOX 1077 HAVERTOWN, PA 19083 (215) 446-0118		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS COMMANDER, ARRADCOM FIRE CONTROL DIV/FC & SCWSL DRDAR-SCF-DA - DOVER, N.J. 07801		12. REPORT DATE April, 1980
		13. NUMBER OF PAGES 196
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) COMMANDER, ARRADCOM STINFO, TSD (DRDAR-TSS) DOVER, N. J. 07801		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

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

18. SUPPLEMENTARY NOTES

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

MICROPROGRAMMED-MICROPROCESSOR
EMULATOR
MASTER INSTRUCTION SET
AUTOMATIC PROGRAM TRANSLATOR
ECONOMIC/TRADE-OFF ANALYSIS
UNIVERSAL PROCESSOR
INTEGRAL PROCESSOR

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

The economic/trade-off analysis for developing common (standard) hardware and common software computer resources for the Advanced Attack Helicopter (AAH-YAH-64) Fire Control System was performed.

The AAH-YAH-64 has a very sophisticated and complex Fire Control System, composed of 14 subsystems which contain 17 micro -

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

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

20. ABSTRACT

processors. The microprocessors are of ten different hardware types and use twelve different assembly languages. The AAH-YAH-64 has been in development for approximately three years. The software documentation is not deliverable under the present contract.

The analysis evaluates three basic approaches toward commonality. These are 1) develop common (standard) hardware computer resources, no change in the software; 2) develop a common software language with software aids for each microprocessor, no change in hardware; 3) develop common (standard) hardware computer resources and a common language with software aids and documentation.

The proposed common microprocessor design uses the American Micro Devices Am29116, 52 pin DIP, microprogrammable microprocessor as an emulator of all the instruction set for approach (1) above. The proposed common software language is the Master Instruction Set (MIS) which maps all of the current instruction sets, and a proposed Automatic Program Translator provides the required software aids for approach (2), and the proposed common microprocessor implementing the MIS in micro-code and the Automatic Program Translator using MIS to generate machine level object code. A ATP is self-documenting. A true High Level Language with an optimizing compiler generating MIS as its object code approach (3).

The analysis examines costs in the area of hardware and software development, maintenance, supply, and training. The proposed hardware and software concept (approach (3)) can be phased into the AAH-YAH-64 program with very little cost and schedule impact. The proposed concept would result in an estimated cost avoidance of over $40 million over the 10 year life cycle. It is estimated that a cost avoidance of $5M to $9.5M could be expected the first year and $1.7M to $3.4M the second year. In addition, the common hardware could result in an estimated cost avoidance of approximately $2.6M in component purchases during production and initial sparing.

The Economic/Trade-Off analysis recommends the implementation of approach (3) as soon as possible.

ACKNOWLEDGEMENT

R. J. Brachman Associates, Inc. wishes to offer their thanks and express appreciation for the cooperation of

Mr. Ron Nippes, District Sales Manager
American Micro Devices
and
Mr. Marvin Sleven, Vice-President
Automated Products Division
Algorex Corporation

for technical data and assistance during the preparation of this report.

We wish to express our appreciation to Ms. Mary Adams and Ms. Jacqueline Zanca for their excellent typing of very complex sections of this report, last but not least, we wish to express our gratitude and many thanks to Doris and her "miracle" workers at D & M Rapid Reproduction, without whose assistance, this report would not have been published.

EXECUTIVE SUMMARY

Introduction (Section I)

The Economic/Trade-off Analysis presented in this report was prepared by R. J. Brachman Associates, Inc., in accordance with the Scope of Work (SOW) Contract DAAK10-79-C-0329. The SOW required the design concept of a common microprocessor and the development concept for a common language and software system to reduce the large number of different MP's and assembly languages present in the AAH Fire Control System. The purpose of the study is to reduce the anticipated very high software maintenance cost expected when the AAH is fielded.

The totality of coverage of the many regulations relating to management of computer resources in military systems causes unintended confusion when applied to microprocessors. The microprocessor is usually employed in a dedicated role, "deeply embedded," physically integrated with other circuitry, and uses applications software with less than 32 K lines of code. It is suggested that the term "Integral Processor" be used with appropriate definitions and boundry conditions to provide clear guidance to Development, Test and Procurement Personnel.

Data Base (GFM) (Section II)

The GFM data base was limited due to the complexities of the Advanced Attack Helicopter Contract and the competition sensitive nature of many subcontracts. The subsystems are identified by letter. The data base used in this study was derived from Reports 79-104 (10) and 79-105 (5) by R. J. Brachman Associates, Inc. These reports compile available hardware and software data derived from questionnaires, direct contact with Prime and subcontractors and commercial sources. The AAH Fire Control System is composed of seventeen different microprocessors implemented in ten different hardware configurations and twelve different languages are used to write the application software. A new microprocessor, the Z-80 was recently added to subsystem K which also contains a 16 bit microprocessor using four 2901A 4 bit devices. The total application software is in excess of 150,000 lines of code.

Technical Approach (Section III)

The present multiple processors and their associated languages were used as the base for comparison of three basic approaches to achieving commonality of hardware and/or software. The three areas explored in this Economic/Trade-off analysis are:

1. Common hardware based upon full emulation of all the instruction sets. This would not affect any of the currently developed software. The detailed discussion of the emulator design is in Appendix A.

2. Common software based upon a common assembly language using a Master Instruction Set, (MIS) described in Appendix C and an Automatic Program Translator described in Appendix B. The Automatic Program Translator is self-documenting and would generate object code for the currently developed MP hardware. This would require 11 code generators. This would not affect any of the current hardware designs.

3. Common hardware based upon microcoding the Master Instruction Set as the common language. The Automatic Program Translator would be the same as 2 above except that only the code generators would be required.

Hardware--A Common Microprocessor (Section IV)

The proposed common microprocessor is based upon the American Micro Devices Am29116 single device microprogrammed microprocessor. The detailed study is contained in Appendix A. A potential problem involving PC board area and power dissipation may exist when attempting to replace the 8 bit microprocessors with the 16 bit Am29116. The recent disclosure that an 8 bit Z-80 MP was added to one of the subsystems may represent the potential solution to the 8 bit packaging problem. Thus, there would be two common MP's, the Z-80 and the Am29116. The repackaging of the current MP hardware designs can be easily handled using design tools such as the Algorex Corporation Automated, Integrated Design and Engineering--"AIDE" (9) system. This system will produce full documentation for production as well as highlighting the design changes, power dissipation MOP, signal tracing between PC boards within the subsystem, logic loading analysis and original to current data mapping as well as other documentation.

Software--A Common Language (Section V)

The proposed common language is based upon a Master Instruction Set (Appendix C) which will permit direct translation between the current 11 assembly language and the MIS. This enhances traceability and would reduce testing of modified software using the MIS. Training, documentation and development costs would be significantly reduced. The existing software would be translated to the MIS using the proposed Automatic Translation of Programs from one computer to another (alternative I). The Automatic Program Translator is also self-documenting. A DoD High Level Language such as ADA could be used to develop an optimizing compiler (Appendix D) which produces MIS as its object code, then MIS would generate microprocessor object code. This procedure actually provides more efficient machine language code (Figure 5).

Economic/Trade-off Analysis (Section VI)

This section consolidates the various cost data presented in other sections. The Economic/Trade-off analysis is summarized by a Matrix, Table I. The implementation of the proposed common hardware/software system will also yield cost avoidance of \$5M to \$9.5M the first year and between \$1.7M to \$3.4M the second year after fielding of AAH. In addition, a cost avoidance of approximately \$2.6M could be achieved due to quantity purchases of MP components during production and initial spares provisioning.

Summary and Recommendations (Section VII)

The proposed common hardware and software system for the AAH Fire Control System is technically feasible and extremely cost effective, even though the AAH has been in development for over 3 years. The proposed program would be non-developmental Product Improvement Program. It is therefore recommended that:

1. The Master Instruction Set be finalized as soon as possible to include all microprocessors.

2. The Automatic Program Translator, System Alternative I be initiated immediately.

3. Initiate the design and brass-boarding of the proposed common MP which implements the MIS as soon as possible.

4. Expand the application of the proposed MP to all fire control applications requiring a microprocessor.

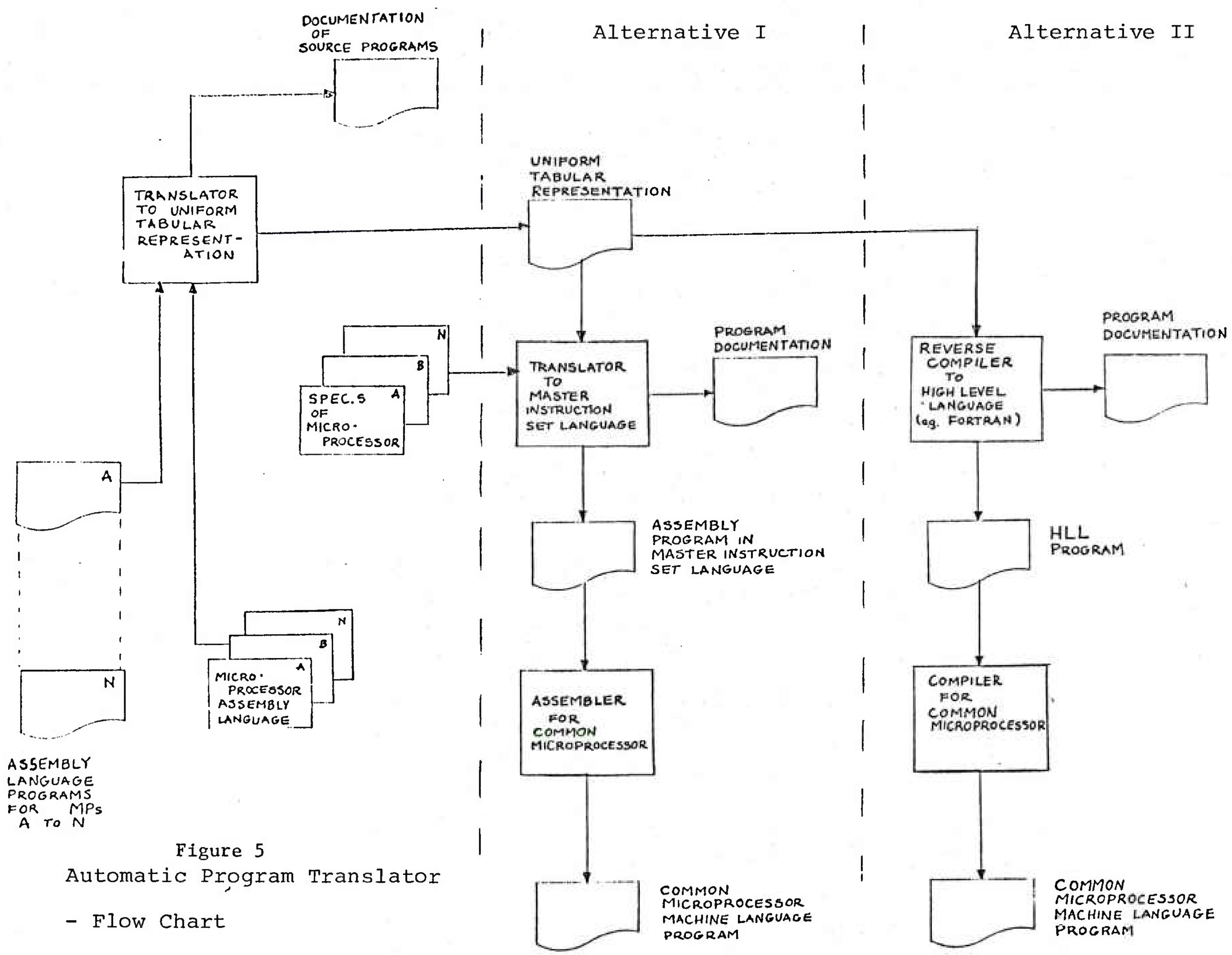

DOCUMENTATION OF SOURCE PROGRAMS
Alternative I
Alternative II
TRANSLATOR TO UNIFORM TABULAR REPRESENT-ATION
UNIFORM TABULAR REPRESENTATION
SPEC.S OF MICRO-PROCESSOR
A
B
N
TRANSLATOR TO MASTER INSTRUCTION SET LANGUAGE
PROGRAM DOCUMENTATION
REVERSE COMPILER TO HIGH LEVEL LANGUAGE (eg. FORTRAN)
PROGRAM DOCUMENTATION
ASSEMBLY PROGRAM IN MASTER INSTRUCTION SET LANGUAGE
HLL PROGRAM
MICRO-PROCESSOR ASSEMBLY LANGUAGE
ASSEMBLER FOR COMMON MICROPROCESSOR
COMPILER FOR COMMON MICROPROCESSOR
ASSEMBLY LANGUAGE PROGRAMS FOR MPs A TO N
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM

Figure 5
Automatic Program Translator
- Flow Chart

Areas of Comparison	Present Multiple Micro-processors, Independent Hardware and Software	Common Micro-Processor Full System Emulator Independent Software	Common Software Language Automatic Translator, Compiler Independent Hardware	Common Micro-Processors Common Soft-ware Automatic Translator Master Instruc-tion Set--Optimizing Compiler
Development				
Common Microprocessor	0	1.2	0	.8
Common Software	0	0	$1.40	.925 (Alt.I)
Software Maintenance	33.8 - 22.9	33.8 - 22.9	25.3 - 17.2	16.9 - 11.7
Software Development System	.52	.52	.03	.03
Program Rewrite	28.4	28.4	22.6	14.2
Software Documentation	.4	.4	.03	.03
TM's/FM's--Hardware	1.3 - .5	.08	1.3 - .5	.08
TM's/FM's--Software	1.9 - .64	1.9 - .64	.12	.12
ATE Software	2.5 - 1.9	.6	2.5 - 1.9	.6
Depot Training	.3 - .18	.04 - .03	.3 - .18	.04 - .03
Supply Line Items Cost	24 - 18	6	24 - 18	6
Total 10 Year Life Cycle Cost	93.1 - 73.4	62.9 - 60.8	77.6 - 62.0	39.7 - 35.0

Table I. Economic/Trade-Off Matrix (Cost in Millions of Dollars)

ASSOCIATES, INC.

TABLE OF CONTENTS

RJB ASSOCIATES, INC.

FIGURES

TABLES

SECTION I

INTRODUCTION

This report is prepared in accordance with the Scope of Work under contract DAAK10-79-C-0329 entitled "An Economic Analysis/Trade-Off Study of all Advanced Attack Helicopter (AAH) System Microprocessors and Associated Devices. The SOW states "this task is to make a detailed analysis/trade-off of all system microprocessors and associated devices and their functions, their physical characteristics, and packaging. A standard microprocessor family will be designed within the system physical and functional constraints. The extent of standardization achievable will be determined.

A detailed analysis will be made of the existing software, adequacy of documentation, special purpose tools, needed for generation of applications software, language requirements, and procedures will be developed for documentation preparation."

Work to be completed ninety (90) days after start of contract.

The AAH has been in development for approximately three years. There are fourteen sub systems under consideration, each having at least one microprocessor (MP). The microprocessors represent seven different physical types of hardware, however they also represent ten different programming languages. The SOW thus represents a very formidable challenge. R. J. Brachman Associates, Inc. project team was able to achieve the unique results of a single common hardware design MP, a master instruction set which will permit the software system to perform automatic code conversion from the present twelve assembly languages to the Master Instruction Set, an optimizing high level language compiler and an automatic documentation generation system. In addition, the recommended approach will be completely cost effective and with early implementation could be phased into the AAH program without affecting the fielding date of the system.

This report is organized as follows:

Purpose of the study.

This will be covered in the introduction and will discuss policy, proliferation of embedded computers, and definitions.

Data Base (GFM)

This will be discussed in detail in Section II. Data Base will cover types and make of processor hardware and software involved and the data available for performing the cost or economic/trade-off analysis.

Technical approach

This will be covered in detail in Section III. Basically the technical approach addresses the three options available when attempting to redesign portions of a given system. These are a) redesign the hardware without affecting the software, b) redesign the software without affecting the hardware, c) redesign both with minimal impact on the overall system.

Hardware, design and packaging

This will be covered in detail in Section IV. This section will cover the packaging of the proposed common MP and the space available based on the information provided by the GFM and analysis of the performance of the sub-system.

Software

This will be covered in detail in Section V. This section will discuss software, how it is handled in the military, how the industry handles it, documentation and its value and the approach proposed for implementation of a common or standard high level language.

Economic/trade-off analysis

This will be covered in detail in Section VI.

This section is the compilation of all the data provided in the other sections and the cost of the various approaches as compared to having the system progress as currently designed.

Summary and recommendations

This will be covered in Section VII. The content of this section is self-explanatory.

A. Purpose of the study

The Advanced Attack Helicopter is a major weapons system having a Project Manager. The program is scheduled for both Army and Department of Defense reviews entitled ASARC and DSARC. During previous reviews it became known that the AAH contained a large number of microprocessors. In addition to the different hardware, a number of different languages are also involved. This caused considerable concern especially in light of DoD policy. DoD policy directed toward reducing proliferation of computer resources and reducing the high cost of computer software (development and in particular maintenance) are well-defined. Automatic Data Processing Regulation (ADPR) covered by the Army Regulation AR18-1 series provides thorough, detailed management procedures in use to implement DoD policy in this area. The management of tactical computers, including embedded computers, acquisition and fielding are to be covered by AR70-XX (draft): Management of Computer Resources in Army Defense Systems. More specific and detailed management policy is covered by DARCOM in its DARCOM Test and Evaluation Guideline (draft) and DARCOM-R 70-16 entitled "Management of Computer Resources in Battlefield Automated Systems." All these and many other supporting regulations use adjectives such as "embedded," "real-time," and/or "closed-loop" when describing both hardware and software computer resources. However when examples are provided, the embedded, real-time, closed-loop computer is substantially large, expensive and is well-definable as a major sub system. The area occupied by embedded, real-time, closed-loop processors which cost several hundred dollars and utilize only several thousand words of programming and are an integral part of the circuitry of the sub-system are not as well-defined and thus many systems and sub-systems will be developed having these microprocessors intermixed throughout the circuity. A means must be established to provide some control and reduce proliferation but not to the prior extent since the cost of implementing the regulation would exceed the cost of the microprocessors by at least an order of magnitude (10 times).

The motivation for producing the policy statement as well as implementing regulations is simply cost. The policy statements of DoD 5000.29(1) and DoD 5000.31(2) specifically state that cost is the basis for the DoD directive/instruction. The complexity of managing computer resources is highlighted by the large number of regulations/instructions covering this area. The problem with implementing a large number of detailed regulations/instructions is the cost and personnel resources. In some current weapon systems, the cost of implementing the regulations/instructions would exceed the cost of developing and fielding the computer resources by at least an order of magnitude. This apparent negative cost ratio should not be a surprise, if one examines the examples and basis for developing the current regulations/instructions. While not specifically addressed, it appears that the processors costing several hundred dollars and requiring one to two thousand lines of instructions does technically come under the regulations/instructions. The cost and human resources required often result in these devices "not being managed." On a case-by-case basis, this may not be a problem. However, a case in point is the current AAH program which has seven different hardware microprocessor configurations and ten different software languages used with these processors.

Today's weapon fire control systems utilize an extensive array of sensing devices in the form of radar, electro-optical, and infra-red devices, laser rangefinder/designators, and arrays on atmospheric, platform, and weapon sensors. Weapons mounted in moving platforms require gyro stabilized platforms with fast response, precision controls as well as control of targeting sensors. The maneuverability required of today's weapon systems require maximum use of the system's physical envelope. Sensors and operating personnel are placed in the most tactically effective position possible. This leads to additonal requirements in the form of data transmission and operational displays. The tactical requirement for combat effectiveness and survivability dictate the need to decentralize the processing of fire control data. Further, there are a number of techniques for processing this data which permit lower cost and more effective use of digital processing technology.

The data processing industry, by nature of the devices used, is a digital industry. Weapon system controls dealing with physical movement of devices and components, operating in a dynamic environment, have been principally an analog industry. The improvement of digital techniques and digital devices has resulted in a transition of weapon systems control and in particular Fire Control to more digital techniques. The end product of almost all fire control functions still is an analog function, i.e., some physical element, a gun, a platform, a sight moved from one position to another. The movement while initially controlled digitally must end up being converted to an analog control signal. The digital technique including microprocessors are totally dedicated to this function. Therefore, the identity of these devices must be defined properly in order to provide their specific place in the management structure of computer resources.

Further, many of the sub-systems involved in a fire control system do not necessarily require processors, much less microprocessors or digital computers. However, after the design and interface requirements have been established, it is generally cost effective to use a microprocessor to reduce the total parts count and number of devices in the sub-system. Therefore, many microprocessors evolve into a sub-system after development has been initiated. Management procedures must recognize this phenomenon.

The current management approach to insure control of a given area of technology or given discipline is to provide all-encompassing and total inclusion of every conceivable facet of that area. In attempting to cover the total spectrum, many control elements, while in the regulation, are actually unmanageable, principally through the lack of proper definition and personnel resources. As an example, the following definitions are extracted from DARCOM-R 70-16:

A-1 ARMY BATTLEFIELD AUTOMATED SYSTEM--A system employing computer resources that operates or has components that operates within the boundaries of the battlefield regardless of the function, mission, or battle involvement. The system may be an offensive, defensive or direct/indirect support system. Examples of such systems are weapons, communications, command and control, intelligence, avionics, missiles, combat support, and combat service support systems.

A-4 COMPUTER--Electronic machinery, which by means of stored instructions and data perform rapid complex calculations or compiles, correlates and selects data. Examples are analog and digital processors, information processors, real-time control processors, electronic calculators, hybrid computers, communication processors and microprocessors.

A-10 COMPUTER RESOURCES--The totality of computer equipment, computer programs, computer data, associated computer documentation, contractual services, personnel and computer supplies.

A-13 COMPUTER SYSTEMS--An interacting assembly consisting of computer equipment, computer programs and computer data.

A-16 EMBEDDED COMPUTER RESOURCES--The totality of computer resources that form a sub-system or part of any Army Battlefield Automated System, e.g., intelligence collection system, target acquisition system, or weapon system. (For the purpose of this regulation the term "embedded computer resource" is replaced by "Army Battlefield Automated System" as defined in paragraph A-1.)

As can be seen from the above definitions, the total all-encompassing nature of the definitions results in an inherent weakness in the real world management of computer resources in Battlefield Automated Systems. A new definition or additional definition is probably not required in view of all those that exist. However, in order to properly associate the microprocessor and its role in the overall Battlefield Automated Systems, a definition more specifically related to this device is required. The following definition is suggested:

A-X INTEGRAL MICROPROCESSORS: An integral microprocessor is the device and its associated components which provides completeness to a sub-system function. It is dedicated in nature and generally does not have the peripherals and internal operating system normally associated with larger computers. The Integral Processor is physically and electronically integrated into the sub-system design and package. It is usually not separable in a physical sense, its role and modifications of its role are dictated by the overall performance of the sub-system within its environment rather than due to outside or external influences.

The Advanced Attack Helicopter program is considered "Competition Sensitive." Therefore the sub-systems are identified by letters. These letters do not have any relationship to the actual function of the sub-system. Throughout this report we will be referring to the various processors and the sub-systems by these letters. This is shown in Figure 1.

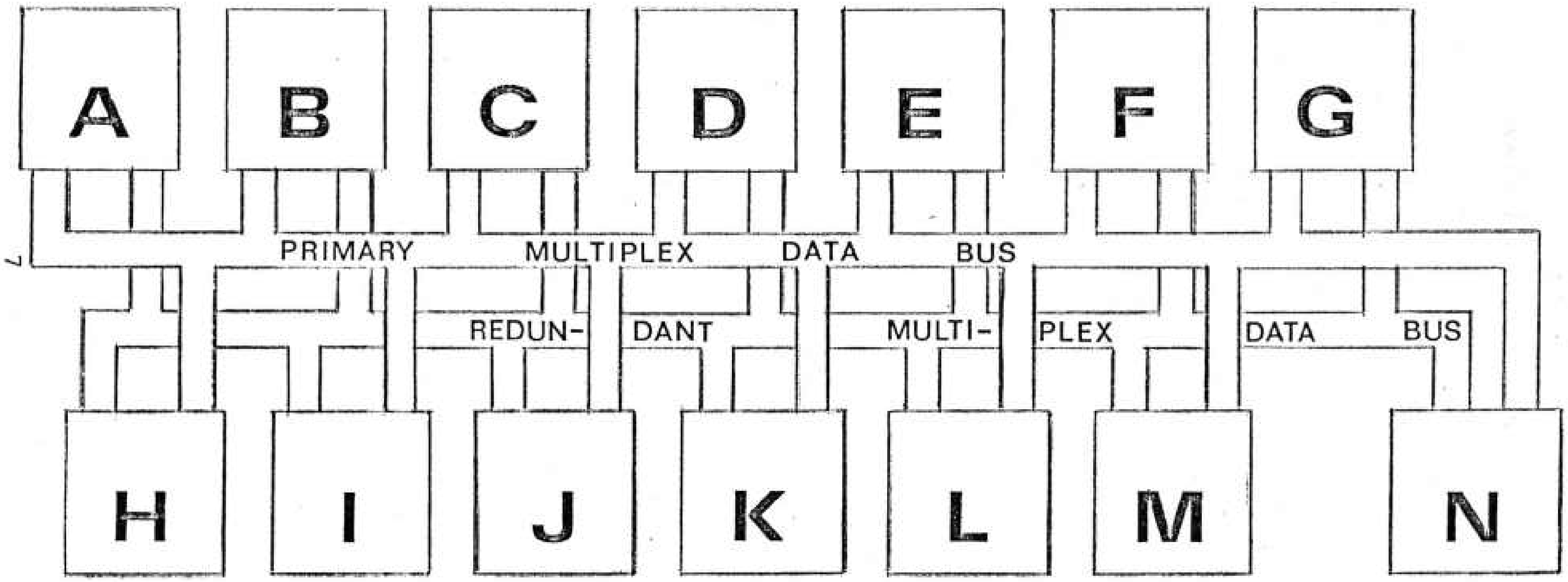


Figure 1

System Block Diagram

SECTION II

DATA BASE

A. The Economic/Trade-off analysis required by the SOW includes a proposed redesign of the microprocessors which will result in a standard or common MP, a common high level language, common software, software aids and documentation system, and the life-cycle cost associated with this effort as compared to the life-cycle cost of the present design. The combination of several factors such as the 14 (or 15) microprocessors used in the same weapon system, the weapon system having been in development for almost three years, and the concept of re-designing the microprocessors and supporting software to achieve the commonality and reduce proliferation makes this study quite unique. The uniqueness of this study effort is further enhanced by the lack of a well-defined and easily acquired data base with which to perform the analysis. The life-cycle cost of a system design is treated differently within the data processing industry and the U.S. Army. The lack of certain technical data relative to the microprocessors also creates a data base problem. The data presented appears to be quite heterogeneous in its composition. This is due to the many varied sources investigated in order to obtain useful data for this study. The data base is not intended to be total or complete but rather to provide sufficient information to support the Economic/Trade-off analysis. In most cases, the data was available from a single source. Where multiple sources provided data and the data differed, the difference was used to provide a "tolerance band." The hardware data is acceptability defined, however, the software data is lacking in a number of areas. Under the current AAH contract, the software and the documentation for the applications program for all but one subsystem is not a deliverable item. Thus the details of the software were lacking. To overcome the lack of detailed data, the algorithms used in the various sub-systems were analyzed and the level of complexity as well as the number of instructions were estimated. This coupled with the data that was available was determined to be sufficient for the purpose of this analysis. The data used in the analysis is described as well as its source.

B. Hardware Data Base

1. Microprocessor Data

The proposed design of a common (standard) MP must solve the most complex as well as the simplest algorithms. Further, the proposed design must be physically packaged within the same unit or sub-system currently mounted in the AAH. The most complex sub-system computational requirement is met by the Fire Control Computer. The specification for this computer is described in CRITICAL ITEM DEVELOPMENT SPECIFICATION FOR FIRE CONTROL COMPUTER, AMC-DC-AAH-H3003B, dated 31 October 1978 (4). This document described the required instruction set and execution times. In addition, it provides a physical envelope for packaging of the computer. Additional hardware data was extracted from the Report 79-105 entitled "Commonality Study of Computer Hardware Resources in the Advanced Attack Helicopter (YAH-64) Fire Control System" (5). This report compiled data contained in questionnaires answered by almost all the Prime Contractor/subcontractors on the AAH.

2. Proposed MP Design

The proposed design of the common microprocessor is based upon the American Micro Devices Am29116. This device is described in a paper entitled "A High Performance 16 Bit Bipolar Microprocessor--The Am29116" (6). Design data and design techniques for microprogrammed microprocessors were obtained from a series of manuals entitled "Build a Microcomputer Distributed by Advanced Micro Devices" (7). These two documents provide excellent design guidance as well as an understanding of the Am29116 microprogrammed microprocessor (MPMP). The supporting devices such as ROM and RAM data were updated using information provided in two parts of a series published in Computer Design, December 1979 and January 1980 (8). This information augments that data presented in the previous report 79-105 (5).

3. Packaging

The physical size of the boards used with the MPs are generally described in report 79-105 (5). The redesign of the boards is not considered a problem with today's technology, as there are a number of automatic circuit design and layout programs available. In particular, Algorex Corporation has a system for Automated, Integrated Design and Engineering called "AIDE" (9) which will permit automatic layout of the PC board along complete documentation complying with military specs such as MIL-STD-275B, MIL-P-55110C and MIL-STD-1495. The Algorex "AIDE" system can provide full manufacturing data and in most cases permit layout of the board without significantly changing most of the components already mounted on the PC board. In addition, special analysis is provided by the Algorex System Map which presents a cumulative analysis of all design changes to insure current up-to-date data in the final drawings. Further, Algorex can generate a Signal Trace Report. This permits the tracing of signals through a number of PC boards. This is important when re-packaging the system and a number of the MP components may be distributed over several PC boards to meet the physical envelope design constraint. This insures a fully operational system prior to being manufactured thus avoiding costly rework.

C. Software

1. Current Microprocessors

The software data available for the current microprocessors is quite limited. Most of the available data has been compiled in a report 79-104 entitled "Commonality Study of Computer Software Resources in the Advanced Attack Helicopter (YAH-64) Fire Control System" (10). The software requirement in the form of solution or algorithm solution times and the number of instructions were deduced from review of two reports published by Hughes Helicopters.

The reports are YAH-64 Phase II advanced Attack Helicopter, Substantiating Technical Data Fire Control Report. These reports are marked "Competition Sensitive." Therefore specific references to the data will not be made in this report, although the information was used to generate software estimates.

2. Development Aids

All sub-systems' software was developed using some development aids. In most cases, the subcontractor utilized the development aids available from the device manufacturer. However, a number of the subcontractors, in particular those using microprogrammed microprocessors, developed their own software aids. If the current design is to be supported by other than the current subcontractors, it would appear that all the development aids would have to be purchased for this purpose. This implies extensive training to be able to utilize the ten different software development systems possible. In developing the cost data, the development aids or systems available from the microprocessor manufacturer will be used. The software aids or development systems vary in cost from $10,000 to $60,000. Custom design software development systems are estimated to be double this cost. All but one sub-system were programmed in assembly language. The one system had 65% of its software programmed in the PL/M (high level) language. The remainder was programmed in assembly language. Some of the development systems produce documentation suitable for a third person to use for maintenance purposes. However, in all cases except one, the documentation produced is proprietary.

3. Development Costs

The cost of developing the current software for the various AAH MP microprocessors is treated as sunk cost. However, the development costs for the new common software will be shown. Determining the maintenance costs per lines of code (or any other unit of measure) is more complex than any other factor in the software maintenance area. A paper presented at a symposium on Computer Software Engineering in 1976 by Gansler (11) indicated the cost to develop software was in the order of $75 per instruction. Another more current paper dated January 1980 (12) indicated the DoD cost of line of executable machine level instruction varies between $40 and $60 per line. Another paper (13) indicates programmer's production capability at approximately 1000 lines of code per year. The figure of $60 per executable line of code would be more comparable to the AAH due to the almost complete use of assembly language programming. A number of software papers attempting to explain development costs, divide the activity into three categories. These are program design, coding, and testing. While most of the papers agree on this breakdown, they disagree on the ratio of effort. For example, a paper published in 1979 (14) shows the ratio of 3:1:3; another paper published in 1973 shows the ratio as 46:20:23, (15); another paper published in 1978 (16) shows the ratio as 40:20:40; a report covering a slightly different area but somewhat related showed the ratio 35:15:45 (17). These variations of cost/time estimates highlight the difficulty in defining the data base. A composite of these numbers will be used to derive the base-line cost estimates for the proposed common microprocessor design of its software. This will be compared to the estimated cost for additional changes of the existing software and an estimated cost to generate similar software using the automatic translator described in Appendix B.

D. Maintenance

1. Hardware

Hardware maintenance in the data processing industry is reasonably straightforward, however, in the military, there are a number of complexities which add significantly to the cost. For example, training and technical manuals become a significant cost when supporting ten different microprocessors as compared to one. The reason ten is indicated here is because the hardware maintenance personnel will require a knowledge of the instruction set and must be taught some software. Military maintenance personnel perform maintenance on the entire subsystem and thus must know how the MP functions as part of the subsystem. Another area to be considered is testing of the microprocessor. The current plan is to provide Automatic Test Equipment (ATE) to support the AAH. Application program for testing a microprocessor and its related components can vary from $80,000 to $200,000 per microprocessor system. Thus it is evident that a single processor is much more cost effective in this area. Other cost areas to be considered are supply pipeline, supply direct exchange items, and maintenance float. The Weapon System availability is another factor which must be considered in the military since the aircraft is of little value if it is not capable of completing its mission. Another maintenance function to be considered is overhaul. All systems go through overhaul at least once during their life cycle. This would require the depots or the overhaul facility to have proper test equipment, training and documentation as well as the material to support the various subsystem microprocessors.

2. Software

Data on software maintenance are more vague than the development costs. Military maintenance personnel are trained to maintain the subsystem. This requires knowledge of the microprocessor and its operational software. While the individual is not permitted to change existing software, that person must be sufficiently knowledgeable as to report back the changes and why they are required if a "software bug" is discovered in the field. Determining the cost of software maintenance is vague in the data processing industry. Most papers on the subject use percentages or ratios of a development cost. For example, one paper (14) quotes an IBM study which states the cost of software modification after the software is fielded is over one hundred times the development cost; another paper (17) indicates that the software maintenance costs 40% higher than the development costs should be expected; another paper (11) indicates the cost of maintaining or modifying a line of code can be as high as $4,000 per instruction. Several other papers indicate the way to reduce the software maintenance costs is to provide the proper documentation during the development phase. The Second Software Life Cycle Management Workshop (13) had as one of its areas of investigation software maintenance. Review of the summary of the findings and results of the workshop showed no discussion of the maintenance problem. Fortunately, the software maintenance cost can be reduced by the same tools that are used to develop the software. Several reports (18) (2) and a paper (19) indicate that maintenance costs can be significantly reduced through the introduction of a number of software aids and tools during the development cycle. In particular, the ability to document the software and to produce object code from documentation represents a significant advantage toward reducing the cost of software maintenance. A paper presented during the workshop (13) entitled "Life Cycle Cost Analysis of Instruction Set Architecture Standardization for Military Computer Base Systems" by Stone and Coleman showed that the GYK-41 (PDP-11) instruction set permitted significant cost reductions in life cycle costs of software. The proposed master instruction set is quite similar to the GYK-41 instruction set. Therefore, considerable cost savings should result.

E. Methodology

The varied nature of the data in the data base and the many sources highlight the problems associated with this Economic/ Trade-off analysis for commonality of hardware and software in a complex weapons system. The section covering the cost analysis will utilize this data base. In each case the numbers developed will be explained as the to source and weighting factor and how it is applied to the Economic/Trade-off analysis.

SECTION III

TECHNICAL APPROACH

A. General

The requirements of the Statement of Work and the short time allocated for this study necessitated R. J. Brachman Associates, Inc. to implement a very direct plan which covers three principal technical approaches to achieve commonality (standardization) of MP hardware and/or software. The three principal technical approaches are:

1. Common Hardware

   This approach considers the redesign of the hardware without changing the software. This would result in reduced supply costs, reduced training costs of maintenance technicians, and possibly reduced personnel requirements as well as other associated cost reductions. The software would not be changed, thus any software problems and related costs would be the same as in the current on-going program.

2. Common High Level Language (HLL)

   The SOW requires the identification of a common HLL, software development tools and a documentation system. The technical approach pursued was to consider the hardware as currently being developed (i.e., 10 software languages) and determine the technology software aids, and documentation system required to achieve the common HLL capability. The software study included investigations into automatic "de-compiling" of the separate assembly languages to the common HLL, then providing a software development system that would be self documenting and produce object code for all of the various MP's. This approach would reduce software life-cycle costs significantly, however, the hardware problems would be the same as the current on-going program.

3. Common Hardware/Software

   This approach was to develop a Master Instruction Set (MIS) capable of solving all the required algorithms and functions of the Fire Control System. The common hardware would implement this MIS and the software would consider translating all the present assembly languages to MIS, then produce the software aids and documentation system for the MIS and a possible HLL. The existing assembly languages would be "mapped" into the MIS, thus showing traceability and reducing the validation costs of the new software. This approach would provide the advantages of both common hardware as well software.

The economics/trade-off analysis of the above approaches is supported by several detailed technical study tasks. The supporting data required to determine the feasibility of developing a common microprocessor, automatic translation of programs, master instruction set, and other software aids, are considered too detailed technically, and of a specific technical nature to be included in the body of this report. Therefore each of these separate studies are included as appendices. Appendix A is the Feasibility Design Study of the Common Microprocessor Hardware. Appendix B is the study of Automatic Translation of Programs from One Computer to Another. Appendix C is the study of the Master Instruction Set. Appendix D is the study of Software Development Aids.

B. Hardware

The hardware aspects of this study cover several areas. These include the physical configuration or packaging of the sub-system and the processor capability for solving the algorithms of the sub systems. Ironically, it is the hardware mechanization of the instruction set that is critical to the solution of the algorithms. Yet the instruction set is involved in the generation of software. This highlights the intimacy between hardware and software in microprocessor and especially in microprogrammable microprocessors (MPMP). A review of the various sub-systems and their associated microprocessors is as follows:

A-6802-8 bit CPU
B, C, I, N-8080A/8085A, 8 bit CPU
D, L-2901A, 4 bit slice microprogrammable microprocessor-16 bits
E-SKC3020, 4 bit slice proprietary CPU-16 bits
F-54LS181, 4 bit slice microprogrammed logic controller-12 bits
G-MECA-43, fire control computer microprogrammed-hybrid technology, 16 bits
H-2901A, 4 bit slice microprogrammed microprocessor-16 bits
J-SBP9900, 16 bits CPU

K-2901A, 4 bit slice microprogrammed microprocessor-
    16 bits
M-SBP9900 (2), 16 bits CPU
O-2901A, 4 bit slice microprogrammed microprocessor-
    16 bits

In summary, there are five 8 bit CPU's, 1 or 3 16 bit CPU's, and 7 microprogrammed microprocessors. The capability of an 8 bit version of microprogram device to solve the 8 bit CPU problems was not in question. The principal concern was ability to solve the fire control computer requirements with the speed required as well as within the total program storage requirements. The fire control computer is the MECA-43 with appropriate input/output capabilities. The back-up fire control computer is a 2901A 16 bit configuration. These two instruction sets were examined in detail, as well as other available instruction sets, also studied. The current technology available to permit emulation of all the microprocessors is microprogramming. A new device to be released the third or fourth quarter of 1980 is the American Micro Devices AM29116. This is a 16 bit microprogrammed device in a single 52 pin DIP package. The device capabilities and the design are discussed in detail in Appendix A. The AM29116 in its single 52 pin DIP package and 100 nanoseconds execution time for a microcode instruction means that there should be very low technical risk in emulating the fire control computer with its hybrid packaging technique and the back up fire control computer using the 2901A. The packaging problems relating to the 8 bit CPU's will be discussed in more detail in Section V. From the initial microcode count and analysis of the MIS, it appears that the 29116 emulation of the fire control computer will permit execution of the application software instructions in the same time or less than the current processors.

Basically, a microprogrammed machine is one in which a coherent sequence of micro-instructions is used to execute various commands required by the machine. If the machine is a computer, each sequence of micro-instructions can be made to execute a computer instruction. All of the little elemental tasks performed by the machine in executing the computer instruction are called micro-instructions. The storage area for these micro-instructions is usually called the microprogram (microcode) memory. A micro-instruction usually has two primary parts. These are: (1) the definition and control of all elemental micro-operations to be carried out and (2) the definition and control of the address of the next micro-instruction to be executed.

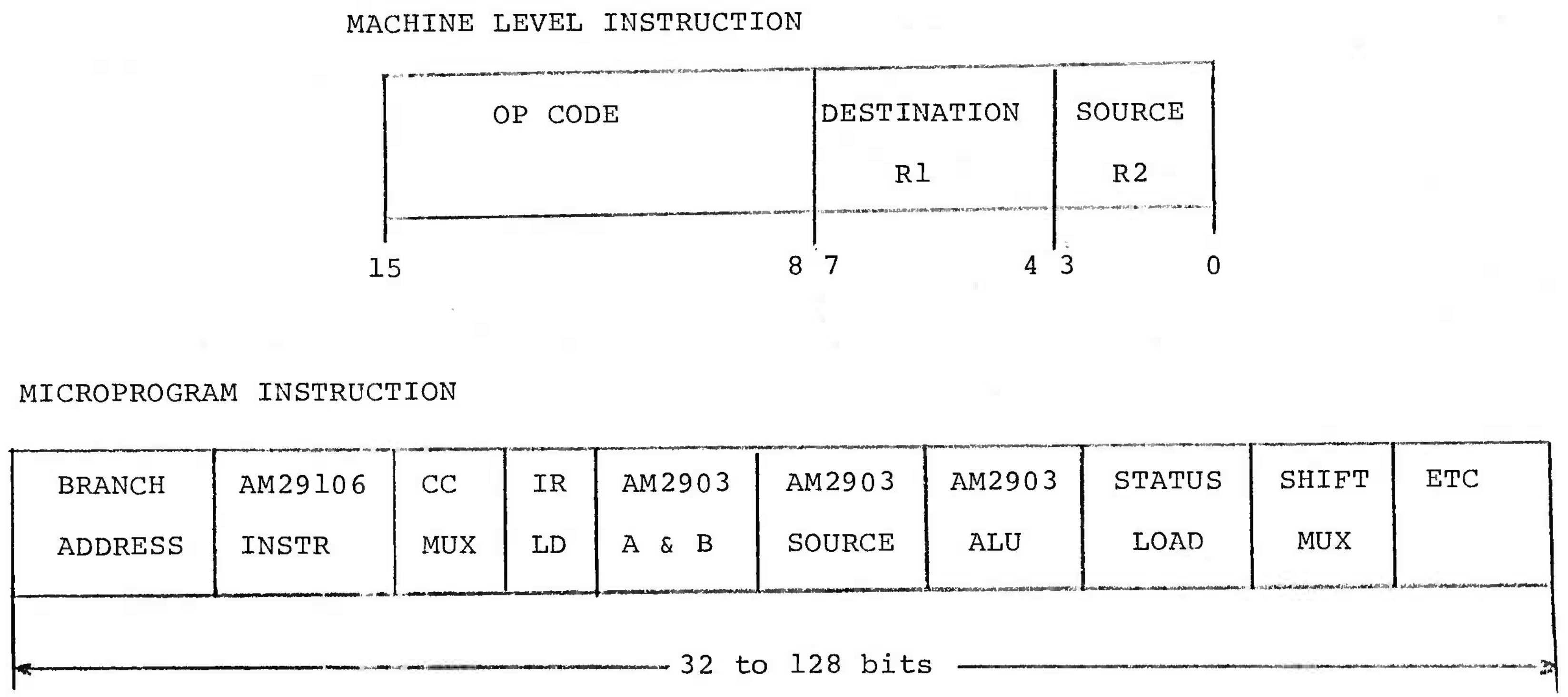
Microprogrammed machines are usually distinguished from non-microprogrammed machines because they are normally considered highly ordered and more organized with regard to the control function field. In its simplest definition, a microprogram control unit consists of the microprogram memory and the structure required to determine the address of its next micro-instruction. Whereas, older, non-microprogrammed machines implemented the control function by using combinations of gates and flip-flops connected in a somewhat random fashion in order to generate the required timing and control signal for the machine. A machine instruction is defined by the number of operational codes to be executed, the number of memory locations to be addressed and the word size of the machine, i.e., 8 bits, 12 bits, 16 bits. In 8 bit machines, if one word will not permit execution of an instruction, then two words must be used. A microprogrammed machine has machine level instructions comparable to the non-microprogrammed machine, however, it also microprograms instructions. These are not dependent upon the work size of the machine level instruction, but on number of control and definition functions to be implemented. Figure 2 is a comparison of machine level instruction and the micro-instruction.

The above description highlights the fact that block diagrams of MPMP's such as shown in Figure 3 and Figure 4 often show the microcode memory as a single block. This block has the same number of input and output lines as the number of bits in the microcode instruction word. The full system shown in Figure 4 and described in Appendix A uses an 80 bit microcode instruction word. The minimal system shown in Figure 3 uses a 56 bit microcode instruction word. A detailed design study (beyond the scope of this contract) should result in a smaller microcode instruction word in both cases.

C. Software

The present MP's in the AAH use ten different assembly languages, depending on the final Fire Control System Configuration. The current software for all the MP's combined amounts to between 150,000 and 200,000 lines of code. The cost to convert this code to a common language would probably equal the original cost to write the code and require several years or a large staff. The initial investigations into the State-of-the-Art of "de-compiling" techniques revealed a few special cases where this had been accomplished. However, further investigations resulted in the conclusion that this approach was beyond the State-of-the-Art and would require extensive research with a very high technical risk. This approach was therefore abandoned.

The next most effective approach investigated was the automatic translation of programs from one computer to another. This proved quite feasible and resulted in several additional



The machine instruction is 16-bits and consists of an op code, source register and destination register specification. The microprogram instruction defines all the elemental signals to control the various pieces of the machine.

Figure 2. Machine level and micro-instruction level instructions

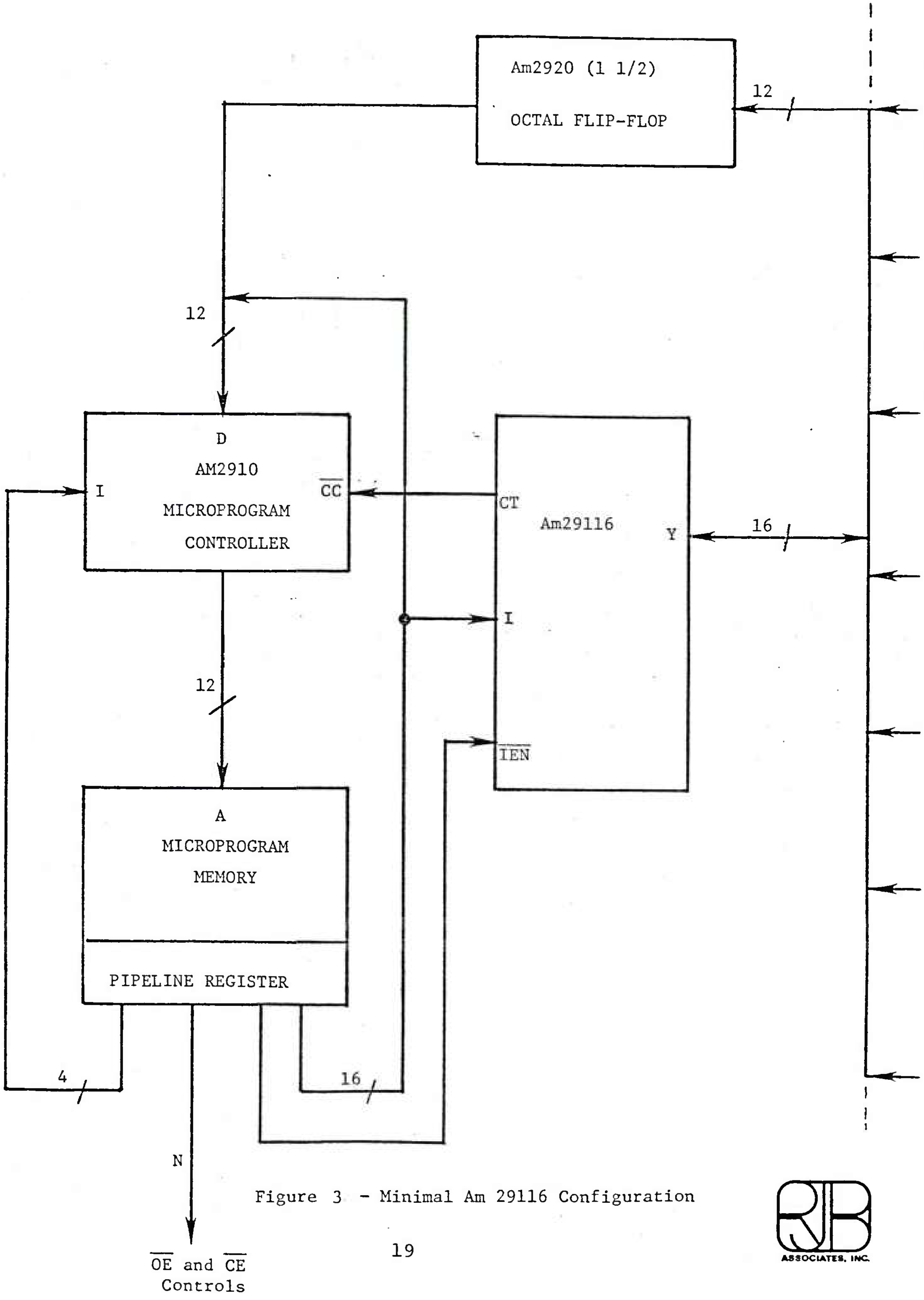
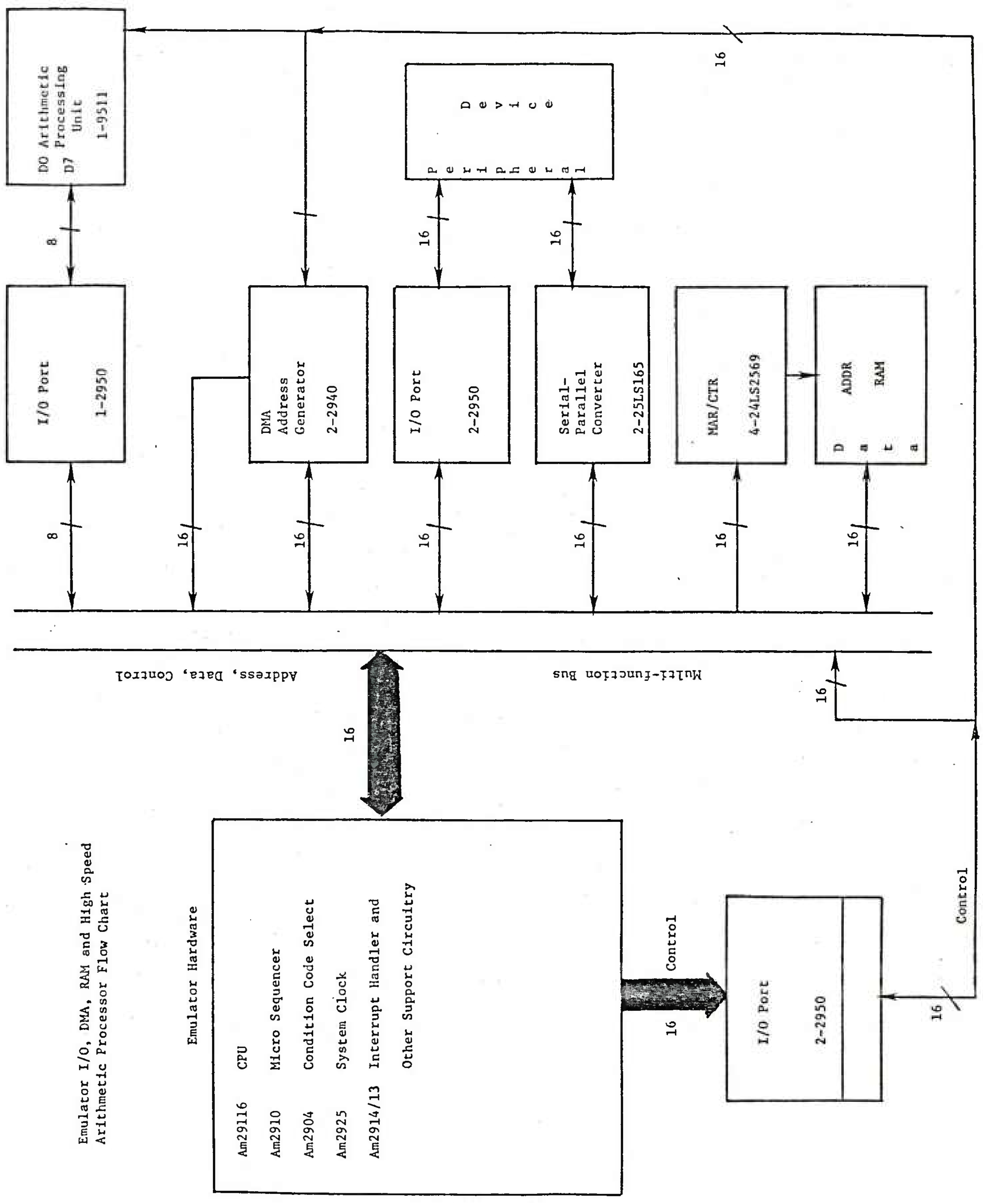


Figure 3 - Minimal Am 29116 Configuration



Emulator I/O, DMA, RAM and High Speed Arithmetic Processor Flow Chart

Figure 4

benefits. In addition, the translation system would be self-documenting.

A common HLL was not specifically considered due to possible availability of ADA. In addition, HLL's have a number of limitations, especially when execution time and memory space are critical. Other limitations were the reason "de-compiling" or reverse compiling was considered not possible. These limitations are described in detail in Appendix B. This does not mean that an efficient HLL and its compiler cannot be provided. The schematic diagram of the translator information flow, figure 5, shows two alternative approaches, these being the MIS or form of MACRO Assembly Language and a HLL with reverse compiler and compiler. Both alternatives are based upon the development of a translator from the source assembly languages of a particular MP into a uniform-tabular-representation of the program. The assembly language translation process analyzes the syntax and local semantics of the individual statements in an assembly language program of any one of the ten source microprocessors and produces a uniform-tabular-representation of the program. It is based upon advanced state-of-the-art syntax analysis techniques which have proved to be invaluable. Specifically, a translator program for these assembly languages will be generated automatically. In addition to checking the statements for syntactic and some semantic errors, the generated program will also store the statements in a tabular form for later processing.

D. Master Instruction Set

The development of a common intermediate or assembly language was pursued due to problems of a HLL and reverse compiling. All the available instruction sets were studied to determine if one could be the candidate common language. The many microprogrammed MP's and the wide variety of instructions led to the conclusion that a Master Instruction Set would be more effective than selecting any one of the MP instruction sets. The MIS described in Appendix C is based upon all the available instruction sets. Subsystems E, H, & K, instructions sets were not available. Further, the entire instruction set of each MP was studied rather than only the instructions used. The resulting MIS provides a very powerful software capability. Thus, it becomes an optimizing common focal point for the development of the MPMP and automatic program translator.

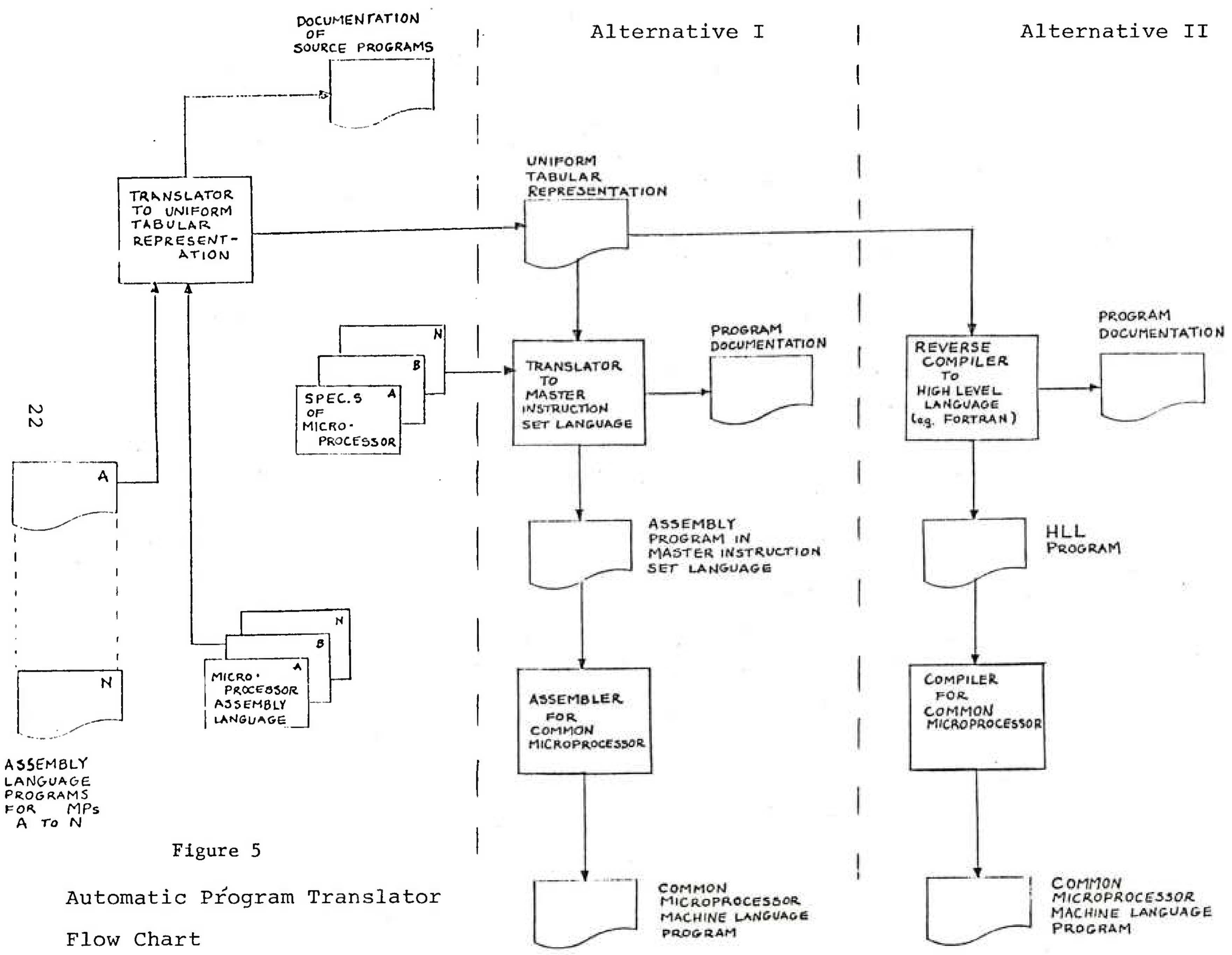

Alternative I
Alternative II
DOCUMENTATION OF SOURCE PROGRAMS
TRANSLATOR TO UNIFORM TABULAR REPRESENT-ATION
UNIFORM TABULAR REPRESENTATION
SPEC.S OF MICRO-PROCESSOR
A
B
N
TRANSLATOR TO MASTER INSTRUCTION SET LANGUAGE
PROGRAM DOCUMENTATION
REVERSE COMPILER TO HIGH LEVEL LANGUAGE (eg. FORTRAN)
PROGRAM DOCUMENTATION
ASSEMBLY PROGRAM IN MASTER INSTRUCTION SET LANGUAGE
HLL PROGRAM
MICRO-PROCESSOR ASSEMBLY LANGUAGE
ASSEMBLY LANGUAGE PROGRAMS FOR MPs A TO N
ASSEMBLER FOR COMMON MICROPROCESSOR
COMPILER FOR COMMON MICROPROCESSOR
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM

Figure 5

Automatic Program Translator

Flow Chart

RJB ASSOCIATES, INC.

E. Software Development Aids

A number of software development aids are discussed in Appendix D. Also included is a discussion of HLL and optimizing compilers.

SECTION IV

HARDWARE-DESIGN PACKAGING

A. General

The Statement of Work requires a packaging and design (redesign) analysis to be performed for the proposed common microprocessor (MP). Redesign of the current PC boards are permitted however, the overall sub-system package cannot be changed. While not specifically stated, the proposed introduction of a common MP for the Fire Control Sub-Systems must not require more total electrical power than is currently provided.

B. Microprocessors, Current Design

The principle source of data relating to the current hardware configurations is Report 79-105, Commonality of Hardware Computer Resources (5). The name "Microprocessor" when used in this Economic/Trade-Off Analysis also includes RAM, ROM, Micro-code memory, and I/O parts. In addition to the mix of microprocessor units (MPU) described in Section III paragraph B, twelve (12) different type RAM devices, eleven (11) different type ROM devices, four (4) different type microprogram sequences (including a proprietary discrete component design), and four (4) different type micro-code memory devices are used in the various sub-systems. The MPU's vary from 40 pin DIP's to 64 pin DIP's plus one MPU configured from four (4) hybrid packages. The memory devices vary from 16 pin DIP's to 24 pin DIP's and are organized from 1024 x 1 bit to 256 x 4 bit devices. Several of the microprocessors have EPROM write circuitry packaged on the PC board. The production version of these PC boards will not contain this circuitry, thus indicating a PC board redesign.

The packaging of the MP's and related components was dictated by the space (volume) available for the particular sub-system in the AAH airframe. The PC boards vary from 4" x 4.5" to 9" x 12" including multilayers and irregular shapes. In addition, the power dissipation requirements, especially the microprogrammed MP's, strongly influenced the PC board and sub-system package design. Unfortunately, MP assembly drawings were not available for this analysis, thus the packaging discussion is somewhat general

C. Common Microprocessors, Proposed Design

The proposed common MP design had to be capable of solving all the sub-system algorithms, from the simplest to most complex, packaged within the available sub-system, and not increase the total power requirements. The proposed common MP design is based upon the American Micro Devices Am29116. This is a 16 bit microprogrammable device packaged in a single 52 pin DIP. The proposed design is described in detail in Appendix A. It will interface directly with all the support devices including the micro-code memory used with the 2901A microprogrammed MP. Thus replacing all the microprogrammed MP's with the Am29116 will not cause any repackaging problems and should reduce the power requirement. The repackaging and power dissipation could be a problem in the sub-system using 8 bit MP's. The minimal configuration (figure 3) represents a processor 30 to 100 times faster (depending on the algorithms used) than the conventional 8 bit MP's. Thus, it represents an "overkill" from the application software standpoint. This, of course, is not a concern if this results in hardware and software commonality. Unfortunately, two potential problem areas may exsist which would negate the use of the minimal configuration in the 8 bit MP sub-systems. These are physical PC board space and power dissipation. The lack of detailed design data including schematics, logic diagrams and assembly drawings is the basis for describing the two problem areas as "potential" problem areas.

The physical PC board space problem area results from the minimal common MP configuration requiring approximately 21 "equivalent units" while the 8 bit MP's vary between 6 and 12 "equivalent units." An "equivalent unit" is an electronic packaging term used to represent the space (area) occupied by one 14/16 pin DIP. The packaging of the proposed common MP is discussed in more detail below. The power dissipation problem area could be sufficiently critical as to require the use of a "second" common MP. The power requirements of the minimal configuration can vary between 3 and 5 times that of the 8 bit MP's to be replaced. This would require redesign of the sub-system power supply which could easily exceed space available within the sub-system. Detailed engineering data is required before a final decision can be made.

D. Design and Packaging

Replacing the current sixteen (16) "Integral" processors in fourteen (14) sub-systems by a common MP appears to be a most formidable task. In fact, opinions such as unrealistic, economically not feasible, and unacceptable delay in fielding the AAH would be expected if it were not for the commonality studies, Reports 79-104 (10) and 79-105 (5), the Am29116, automated design and packaging techniques such as the ALGOREX AIDE^R, and this Economic/Trade-Off Analysis.

The use of automated PC board design and packaging techniques during R & D is generally accepted. There are many different design systems available today. The quantity and quality of documentation provided by these automated systems varies from very little to comprehensive. However, most of these systems are not suitable for design modifications after the design has been released(accepted), nor are they suitable for redesigning sections of the PC board while the other components remain fixed in their original positions. An automated system meeting the requirements for design and re-packaging the MP PC boards in the Fire Control Sub-Systems is the ALGOREX AIDE^R (Automated Integrated Design and Engineering). AIDE can accept raw logic diagrams, schematics or equations and produce the bulk of the drawings, artwork, NC tapes and other required documentation. The ALGOREX AIDE^R automatically checks the design and provides engineering diagnostics, partitions the system, if not specified, provides optimum assignment and placement of components, if not specified, generates assembly drawings, provides routing data between PC boards or hybrid LSI's, produces photo-ready artwork for manufacturing, provides drill templates and/or control tapes for automatic drilling machines, provides control tapes for automatic component insection and resting machines, generates punched tapes for a wide variety of numerically controlled machining operations (APT), and designs wired back-panels, fully methodized wiring process sheets, or control media for automatic or semi-automatic wiring machines. In addition, it generates documentation for engineering, de-bugging, publications, and field service such as Signal Code List, Reference Designation and Pin List, Signal Description List, Thermal Map, Temperature Map, Power Dissipation Map, Original to current Data Mapping, Cumulative System Analysis Map and a Signal Trace Report. The drawings comply with military specifications such as MIL-STD-275D, MIL-P-55110C and MIL-STD-1495. Utilization of the ALGOREX AIDE^R would result in the first redesigned PC board becoming available for component population and test in 3-4 months after start. The entire redesign could be completed in 12-18 months depending upon the available design data.

The several 8 bit MP sub-systems will require component and software analysis to redesign. The software analysis is required to determine if a memory capacity vs. speed trade-off can be made. Many application programs written for 8 bit MP's use tight in-line coding to meet the solution time of the algorithm. This is accomplished at the cost of additional memory. The actual dollar cost is low-due to advances in memory technology thus making this methodology acceptable. Utilizing the speed of the proposed common MP, memory requirements may be reduced between 20% and 50%, thus increasing the probability of replacing the 8 bit MP's and their supporting devices. The only other problem area not covered in the Economis/Trade-Off Analysis is the power requirements. The only comment possible in this area without analysis of the current sub-system design is that the overall power requirements for the proposed common MP will be less than the current requirements.

E. The 17th Microprocessor

This paragraph was added after the final draft of this Report was submitted for review and comment. Information was provided about mid-March, 1980, that sub-system K had added a Z-80, 8 bit microporcessor to the 2901A-16 bit MP already in the sub-system. This disclosure highlights comments relative to management of "Integral Processors" in Section I, Introduction. This late disclosure prevented the Z-80 from being discussed in most of this report. Section VI was partially modified to account for a worst case solution requiring two common MP's. The two MP's would be the Am29116 to replace all 16 bit and/or micro-programmed MP's and the Z-80 to replace all 8 bit MP's. The Z-80 will execute the instruction set of the 8080A/8085A thus minimizing the impact upon software maintenance.

SECTION V

SOFTWARE - A COMMON LANGUAGE

A. General

This section discusses the areas of software costs, as well as application software, development systems, training (Development and Field Maintenance), documentation, and maintenance. The literature relating to large software systems is voluminous. However, the literature devoted to MP software is very limited. None of the literature specifically considers maintenance (some allege to) life cycle costs, multiple processor systems using different processors, or software production aids. This lack of other source data further highlights the originality of this study.

B. Common Software

The keystone of any common software system is its language. Traditionally, reference to a "common software language" implied a High Level Language. A number of MP companies use a common assembly language for a "family" of devices. However, the commonality generally was upward. As the MP's became more powerful, even this form of commonality was lost. In order to avoid costly software rewrites and maintain user confidence, the MP companies developed "cross-assemblers." This is software used to translate one language to another (more powerful to less powerful). The efficiency of the cross-assembled software varied greatly in solution speed and memory requirements when compared to manually programming each different MP. This fact did not appear to affect commercial applications of MP's. However, Military weapon systems using "Integral" MP's could not tolerate these inefficiencies. In many weapon systems microprogrammed MP's, with a unique (problem oriented) instruction set, are used to meet the system performance requirements. This is highlighted in the Fire Control System of the AAH. Seven subsystems which solve very complex algorithms and/or have stringent thruput time requirements use MPMP's with unique instruction sets.

Analysis of the available instruction sets of the MP's resulted in the disclosure that none of the instruction sets had adequate addressing modes to qualify as the common assembly language. A Master Instruction Set (MIS) was designed to support the FCS MP's. A detailed task report describing the MIS and showing the relationship and/or mapping of the available MP instruction sets is contained in Appendix C. The mapping of the instructions sets into the MIS will enhance the traceability of the translated software. The instruction sets of three (3) MPMP's were not available during this study. Therefore, approximately 4 man-months would be required to finalize the MIS.

The proposed use of the MIS is not in lieu of a HLL, but rather as part of a two level software development capability. The HLL finally selected would compile to the MIS. The MIS would then be used to generate the MP Object Code. This approach overcomes the problems (Appendix B) of using a HLL compiler to generate the efficient object code required by the "Integral" MP's.

C. Automatic Translation of Programs

Assembly language programs have been treated as special cases in the software world. This has principally been due limited documentation and most of all the idiosyncrasies of the original programmer. Most software engineers agree that modification of an assembly language program by a "third person" entails a high technical risk and an associated very high cost. Many times the "third person" can show it would cost less to rewrite the program rather than try to modify it. The AAH Fire Control Subsystem MMP's currently utilize ten (10) different assembly languages. Thus, it becomes obvious why there is considerable concern as to the potentially very high software support costs. Discussions relative to development of a common language are contained in subparagraph B above and Appendix B and C. The two software options include: a) using the current assembly languages, a common assembly language and generating the object code using cross-assemblers and b) using a common MP with its assembly language and rewriting the existing software. The cost to completely rewrite of all the software manually is considered close to the original R & D software costs. Realistically, there are many more variable and unknowns during the R & D phase. We estimate that the manual rewrite would cost between 50% and 60% of the R & D costs. In either case, the cost and time to manually develop a software system using a common language would probably deter its implementation. Fortunately, R. J. Brachman Associates, Inc.'s Task Study Group developed a technique for Automatic Translation of a Program from one computer to another. A schematic diagram of the proposed approach to Automatic Translation is shown in Figure 5. It should be noted that each step produces complete documentation and the end product, object code, is produced from the documentation. Development of the Automatic Translator, including translation of all existing

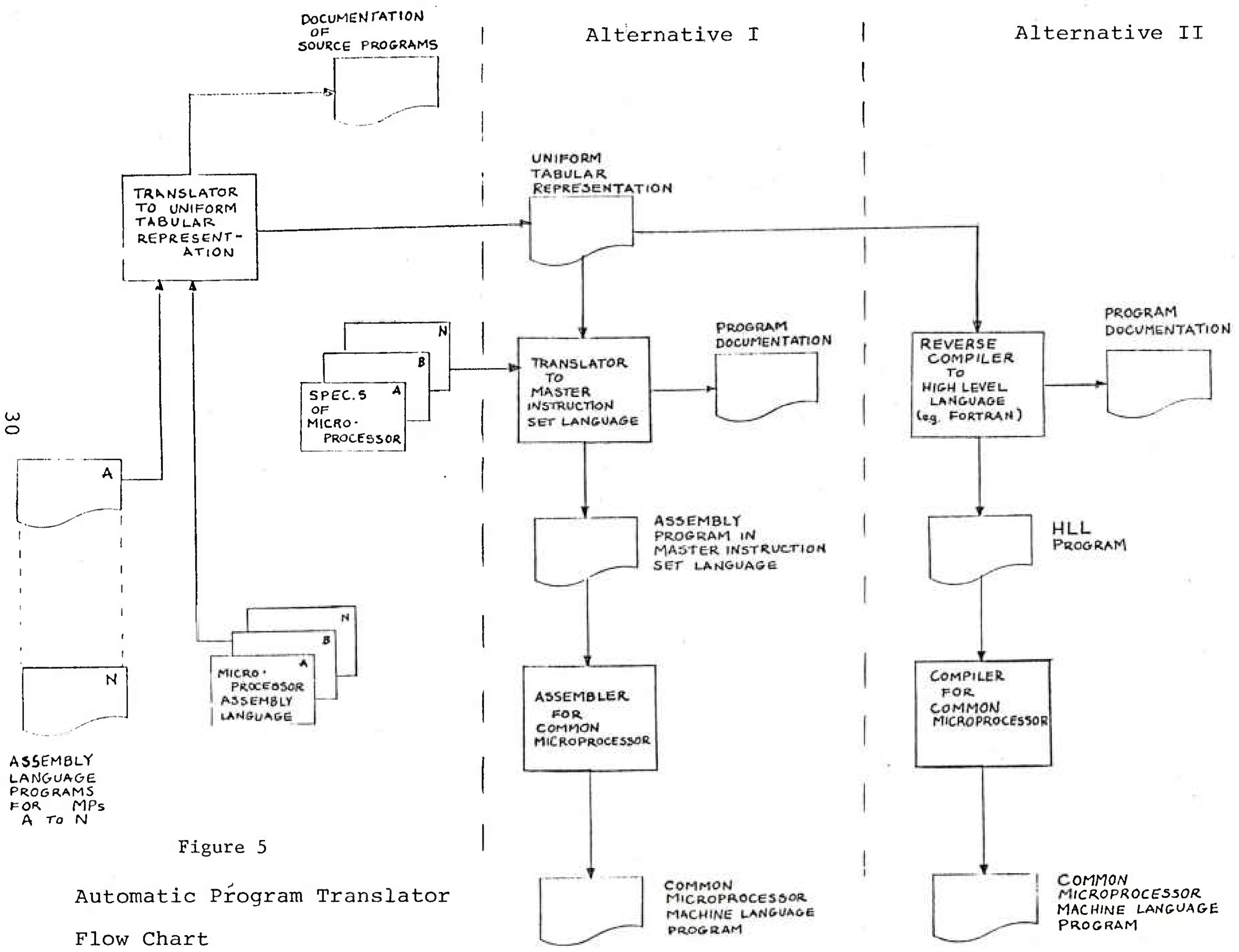

DOCUMENTATION OF SOURCE PROGRAMS
Alternative I
Alternative II
TRANSLATOR TO UNIFORM TABULAR REPRESENT-ATION
UNIFORM TABULAR REPRESENTATION
SPEC.S OF MICRO-PROCESSOR
A
B
N
TRANSLATOR TO MASTER INSTRUCTION SET LANGUAGE
PROGRAM DOCUMENTATION
REVERSE COMPILER TO HIGH LEVEL LANGUAGE (eg. FORTRAN)
PROGRAM DOCUMENTATION
ASSEMBLY PROGRAM IN MASTER INSTRUCTION SET LANGUAGE
HLL PROGRAM
MICRO-PROCESSOR ASSEMBLY LANGUAGE
ASSEMBLER FOR COMMON MICROPROCESSOR
COMPILER FOR COMMON MICROPROCESSOR
ASSEMBLY LANGUAGE PROGRAMS FOR MPs A TO N
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM

Figure 5

Automatic Program Translator

Flow Chart

programs to MIS (Alternative I) is estimated to require 5 man-years over a 1 1/2 calendar year period plus computer time. Implementation of the Automatic Translator using the MIS and generating object code for the current MP's would require a code generator for each assembly language. Thus, Ten Code generators would be required. Design of each code generator requires approximately 6 man-months.

D. Software Development Costs

The cost of software development for the present MP's is treated as a "sunk" cost. The cost to develop the proposed software system is included in the overall Economic/trade-off Analysis.

E. Software Development Systems

All the current application software was developed using a commercial development system or a custom designed system. Commercial development systems cost between $10,000 and $50,000. Custom systems are estimated to cost twice the above. In addition, the custom systems are considered proprietary. This is not critical since commercial development systems are available for all MP's used in the AAH. The number of development systems required to support the AAH FCS will depend upon how many different facilities will support the software. The development system for the proposed common MP/MIS is estimated to cost $30,000.

F. Training

Training requirements for MP support cover many areas. The principal areas considered for the Economic/Trade-off Analysis are as follows:

1. Assembly Language Programmers

The transition from R & D to production may or may not involve the same subcontractors. In either case, it is assumed that new programmers will be provided for production and field support. The number of programmers are estimated to vary from 6 to 28 depending upon the number of different subsystems, subcontractors, and languages in use. This is a conservative estimate since the number of programmers during the R & D phase varied between 52 and 90. Generally, it requires 3 months for a trained programmer to become proficient in a given language. For the purpose of this analysis, we will consider

14 subcontractors, each with 2 programmers using the present multiple MP's and one facility having 6 programmers and using the MIS/Automatic Translation System.

2. Software Development Systems

The above programmer personnel will be required to use an appropriate software development system. Suppliers of these systems estimate it would require between 45 and 60 days of continuous use to become proficient. The cost impact is directly proportional to the number of languages supported and the number of programmers.

3. Field Maintenance

a. Microprocessor Testing

Field maintenance is currently planned to be performed by use of Built-in Test Equipment (BITE) at the Aviation Unit Maintenance (AVUM) level and use of Automatic Test Equipment (ATE) at the Aviation Intermediate Maintenance (AVIM) level and depot. The current maintenance concept does not require Field Maintenance Technicians (FMT) to be trained in MP logic and software since subassemblies will be replaced at the AVUM and PC boards at the AVIM. Unfortunately, the maintenance concept may be unrealistic at the AVIM due to the high software cost and thus limited diagnostic capability of the ATE. MP's and their associated components are complex devices to test. Assuming the ATE can achieve an acceptable level of PC board level diagnostics, the functional test software for the MP and its associated devices, could vary between $20,000 to $60,000. However, the PC boards and subassemblies must be repaired at the Depot. This requires the ATE at the Depot to fault isolate to the piece-part level. ATE software (and hardware) costs can vary between $80,000 and $400,000 depending upon the test accessibility and degree of diagnostics achieved. These cost estimates are supported by previous studies (18) (20) relating to ATE software. In addition, it is assumed that the cost of the Depot Maintenance Work Requirement (DMWR) is included.

b. Technical Manuals (TM's)/Field Manuals (FM's)

TM's and FM's are required for all fielded systems. Even though BITE and ATE are supposed to reduce the technical skill levels in the Field and Depot, FMT's and organization personnel will still be required to have some knowledge of the subsystem operation. The TM's and FM's will have to contain functional descriptions of the hardware as well as the software. It is estimated that the hardware descriptions can vary between 80 and 200 pages and the software descriptions can vary between 100 and 300 pages. The cost per page to prepare these manuals varies between $150 and $225. Each subsystem will require its own TM and FM. Thus, even though a number of MP's are the same, each different subcontractor will prepare a different TM and FM. A common MP and a common software language would result in the same data appearing in all the TM's and FM's.

G. Software Documentation

Current information provided by the PM's staff indicates that the software documentation from only one subsystem is deliverable. All the other subsystem contractors consider their documentation proprietary. A number of papers (16) (17) (18) show that the quality of the documentation directly affects the cost of software maintenance. A programmer's manual is required in addition to the application software documentation. The conventional MP programmer's manuals can be obtained from the manufacturer, however, programmer's manuals for the seven MP's which are microprogrammed must be obtained from the subsystem contractor. These manuals are estimated to cost between $20,000 and $30,000 each. There is some question as to whether the U.S. Army will purchase any of the documentation or "wait" until the Production Phase. It is estimated that the cost to purchase the software documentation will cost about the same as the Alternative I Automatic Program Translator (ATP). The ATP should significantly reduce the cost of documentation during the Production Phase.

H. Software Maintenance Cost

Several papers (11) (12) (14) have been written on the subject. However, sections of the papers have to be combined to provide useful information. The paper by Gansler (11) quotes a U.S. Air Force study showing the cost of Software Maintenance can be as high as $4,000 per line. The paper by Schindler (12) states that DoD expects the cost per line of executable machine-level code to rise from $40 per line to $65 per line by 1984.

Most HLL compilers produce 8 to 20 lines of machine-level executable code per HLL statement. Another paper by Schindler (14) quotes an IBM study which states that the cost to modify a line of code after the software has been fielded is 100 times the development cost. Combining data from both Schindler papers (12) (14), the $4,000 per line of code maintenance cost stated in the Gansler paper (11) does not appear to be unreasonable. It is estimated that $4,000 represents between two and three man-weeks of effort.

Military weapon systems are tested during the R & D and Production Phases. However, these tests only approximate the tactical operation environmental. Thus, these sytems generally require a number of changes during the first two years in the field. Changes/modifications to fielded U.S. Army weapon systems are via Engineering Change Proposals (ECP). Experience indicates a system as complex as the AAH could have 100 to 200 ECP's per month, the first year, 75 to 150 ECP's per month, the second year and approximately 50 ECP's per month throughout its life cycle. It is estimated that 25% of the ECP's, 1st year, 15% ECP's, 2nd year and 8% ECP's, throughout the life cycle, will result in software changes. This results in an estimated 300-600 software changes, 1st year; 135-270 changes 2nd year; and 50 changes per year throughout the life cycle. The first year in the field should produce the most extensive changes. It is not unreasonable to expect that each software change will average 25 lines of code during the life cycle. This results in an estimated 7,500-15,000 lines of code 1st year, the second year 3,375-6,750 and 1,500 lines of code throughout the life cycle. Estimated costs could vary between $30 million and $60 million the first year to approximately $6 million per year throughout the life cycle. Considering a cost as low as $1.000 per line, the cost can vary between $7.5 million and $15 million the first year to $1.5 throughout the life cycle.* A study by Stone and Coleman (13) shows that the Instruction Set Architecture can have a significant impact on the cost of software maintenance. The proposed MIS is very similar to the Instruction Set described as resulting 49% lower maintenance cost as compared to other military computer instruction sets. Solutions to the high cost of software maintenance are being pursued by many organizations. A paper by Goetz (19) provides "steps toward solution" of the high cost of software maintenance. The proposed MIS, Automatic Program Translator, Alternative I, the documentation system and an optimizing compiler for the HLL coincide with the "steps toward solution." It is estimated that the proposed software systems could reduce software maintenance

*Considering the above represents changes in lines of code from 10% the 1st year to 1% throughout the life cycle, the estimate appears to be reasonable.

costs by 50%.

Another area of maintenance unique to the military is Overhaul. During this activity, the weapon system is completely rebuilt so it is the equivalent of a new system. Software also is Overhauled, although the term "program rewrite" is used to describe this activity. This activity is somewhat random as to its occurrence. A weapon system undergoes many changes during its life cycle. These changes may affect the software. Further, weapons, subsystems and tactics will change. Experience has shown that a weapon system such as the AAH could have two complete program rewrites during its life cycle. The number of personnel involved with the system rewrite will depend upon the number of different MP's, subcontractors, and different assembly languages being used. The level of effort is estimated to vary between 22 and 150 man-years. Thus, the two rewrites would required between 44 and 300 man-years of effort. The range of personnel to perform the rewrite is derived from data presented by Putnam (13), Thibodeau and Dobson (13) and Parr (13).

SECTION VI

ECONOMIC/TRADE-OFF ANALYSIS

A. General

This section consolidates the data presented in the other sections of this report. The final cost summary will contain estimated MP hardware and software development and support for the current AAH, the Common MP (no software changes), Common Software (no hardware changes), and Common Hardware and Software based upon the MIS and the Automatic Program Translator.

B. Current AAH MP Life Cycle Cost Estimates

1. Hardware

The data presented in report 79-105 (5) indicates that the MP's would add 300-400 new line items to the supply system. At an extimated cost of $6,000 per line item, this becomes $1.8 million to $2.4 million per year, or $18 million to $24 million over the 10 year life cycle. (This does not include cost of the parts.) One MP using the 54 LS181 logic controller will probably have to be replaced due to impending obsolescence. The redesign should cost approximately $200,000-$300,000. The MECA-43 uses hybrid packages which are proprietary. In addition, the Doppler system uses a proprietary bit-slice MP. These should be replaced to maintain ease of replacement, and supply. Thru competitive procurement current MP technology as shown in the proposed common MP design could easily replace these proprietary devices with a significant cost avoidance during the AAH life cycle. The cost summary does not include the cost of redesigning the MP using the 54 LS181.

2. Technical Manuals and Field Manuals

The MP hardware sections of the TM's and FM's are estimated to require between 80 and 200 pages each. At an average cost of $200 per page, this becomes $32,000 to $80,000 per MP or $512,000 to $1,280,000 for the 16 microprocessors.

3. Training--Depot Personnel

The MP's will be repaired at the Depot. Even though it is planned to use ATE, the repair technicians will require training for each of the MP's and how they function in the subsystem.

The training cost includes the course as well as the technician's salary. Based upon data compiled in the ATSS Economic/Trade-off study (20), six to ten technicians over the life cycle will be trained for six weeks on each MP. This is 288 to 480 man-weeks @ $16.00/hr.= $184,320 to $307,200.

4. Automatic Test Equipment (ATE)

Automatic Test Equipment will be used in the field at the AVIM and at the Depot. The ATE used at the Depot will fault isolate to the piece-part level. It is not known whether the AVIM ATE software will be a subset of the Depot software. For this analysis, the AVIM ATE software will be considered a subset of the Depot ATE software. The degree or amount of "probing" (manually touching a test point with a probe under ATE direction) to be used in testing the MP's is unknown. Extensive probing can double the cost of the ATE software. The Fire Control Computer has the most complex testing requirements whereas subsystem N using an 8085A MP probably has the least complex testing requirements. ATE software costs discussed in ATSS Economic Study (20) were as high as $700,000 for a mini-computer not much more complex than the Fire Control Computer used in the AAH.

Prior ATE software estimates based upon (18) (20) provided the range of $80,000 to $400,000 for MP's. The complexity of the Fire Control Computer places it at the top of this range. It is estimated that the ATE software for the FC computer could easily exceed $400,000. For this analysis, estimated ATE test software costs for the MP's in each subsystem are as follows:

A @ $80,000
B, C, I, & N @ $80,000
J & M (2) @ $95,000
E, F, & H @ $180,000
D, L, & O @ $250,000
K, (2 different MP's)@ $260,000
G @ $400,000

The total ATE software costs can vary by a large amount, depending upon whether one company develops all the software or whether each subcontractor develops the software for their own subsystem. For this analysis, the ATE software will be developed by one company. Additional cost complexities arise due to the unknown level of testability. For example, Subsystem B has the MP and memory plus I/O mounted on two PC boards with excellent test accessibility, whereas subsystem I has the MP, memory and I/O integrated with the other electronic components on the same PC board thus providing poor test accessibility. The probing requirements could easily double the software costs of I as compared to B even though most of the MP test software is identical. The ATE software cost estimates are based upon each MP test program being developed separately and all common MP test programs being developed together. The common development should result in 50% less software costs for each additional MP of the same type. The ATE software cost estimates for the MP's by subsystem then become:

Individual Software (K=$1,000)	Common Software
A = 80K	A = 80K
B, C, I & N = 320K	B, C, I & N = 200K
J & M (2) = 190K	J & M (2) = 145K
E, F & H = 560K	E, F & H = 360K
D, L & O = 750K	D, L & O = 500K
K (2 different MP's) = 260K	K (2 different MP's) = 260K
G = 400K	G = 400K
$2,540K	$1,945K

Note: The above ATE software costs are estimated for the microprocessors, memory and related I/O only. ATE software costs to test the entire subsystem, PC board, or assembly will be more extensive and are beyond the scope of this study. Subsystem K has both a 2901A, 16 bit MP and a Z 80, 8 bit MP. Subsystem M has 2 identical MP's.

5. Software Development Systems

Software Development Systems are discussed in Section V, Para. E. For the purposes of this analysis, the average cost of a system for conventional MP's is estimated to be $20,000. Therefore, six subsystems developed by six subcontractors would cost $120,000. Custom development systems are estimated to cost $40,000. Therefore, ten subsystems developed by ten subcontractors are estimated to cost $400,000. The development system for the common MP design is estimated to cost $30,000.

6. Software Maintenance

a. Experience has shown that the AAH software will be completely rewritten twice over the projected 10 year life cycle of the system. These changes may be caused by major changes in subsystems, weapons and ammo, and tactical use of the AAH. The level of effort for this activity can vary between 44 and 300 man-years for the two rewrites. This results from using methods for estimating software costs discussed in papers referenced in Para. H, Section V. Thus, it is estimated that approximately 56 man-years per rewrite would be required if each subcontractor made the rewrite individually. A central organization performing the rewrites would reduce this to approximately 35 man-years per rewrite and a central organization using the proposed MIS and Automatic Program Translator would reduce the level to 22 man-years per rewrite. The two program rewrites are estimated as follows:

Individual Rewrite	112 M-years @ $100,000 = $11.2 million
Central Rewrite with Automatic Code Generators	70 M-years @ $100,000 = $ 7 million
Central Rewrite using Automatic Translator and MIS Language	44 M-years @ $100,000 = $ 4.4 million

b. Software maintenance costs based upon ECP's (Section V) and $1,000 per line of code are estimated as follows:

(Costs in Millions of Dollars)

	Individual Maintenance Present Software Present Hardware	Central Maintenance Common Software Present Hardware	Central Maintenance Common Software Common Hardware
1st yr. (avg.)	7.5 - 1.5	5.6 - 11.2	4 - 7.5
2nd yr. (avg.)	3.4 - 6.8	2.6 - 5.1	1.7 - 3.4
Each yr. (avg.) × 8	12	9	6
Total 10 yr. ECP Life Cycle Cost:	22.9 - 33.8	17.2 - 25.3	11.7 - 16.9

The common software, present hardware would require the development of the Automatic Program Translator and 11 separate code generators in addition to and development of an optimizing compiler completion of the MIS. This effort is estimated to cost:

APT	$ 450,000
11 Code Generators (11 × $36,000/code)	396,000
MIS	50,000
Compiler	500,000
	$1,396,000 = $1.4 million

C. Proposed Common MP Life Cycle Cost Estimate

1. Hardware

a. Full System Emulator

The design described in Appendix A indicates that development of the microcode for the six MP's analyzed would cost approximately $350,000. Using a conservative cost estimate, the three additional MP's would cost $150,000. The development and debugging of the hardware is estimated to cost $200,000. Therefore, the development costs for the Full System Emulator is estimated to be $700,000. The proposed design would add approximately 100 new line items to the supply system. Thus, at an estimated cost of $6,000 per line item, the supply system costs become $600,000 per year or $6 million over the 10 year life cycle.

b. Master Instruction Set Implementation

The design of the common MP using microcode to implement the MIS would be similar to the Full System Emulator. The principal difference is the reduced amount of microcode. Thus, the implementation of the MIS and common MP is estimated to be $200,000 for the hardware design, $50,000 for the MIS microcode and $100,000 development costs for a total cost of $350,000. The number of line items introduced into the supply system is the same as the Full System Emulator. Thus, the cost is estimated to be $600,000 per year.

2. Technical Manuals and Field Manuals

The MP software sections of the TM's and FM's are estimated to require between 100 and 300 pages each. At an average cost of $200 per page, this becomes $40,000 to $120,000 (per TM and FM).

3. Training-Depot Personnel

The MP's will be repaired at the Depot. Even though it is planned to use ATE, the repair technicians will require training on the MP's and how they function in the subsystem. The use of a common MP and MIS should make the technicians more proficient. Based upon data compiled in the ATSS Economics Study (20), six to ten technicians would be trained for six weeks over the life cycle. This then becomes 36 to 60 man weeks @ $16.00/hour or $23,000 to $38,400.

4. Automatic Test Equipment

From the discussion in Para. B-4 above, it is assumed that the proposed common MP design would be as complex as the Fire Control Computer (subsystems D, L & O are a more realistic comparison) to test. This, then, results in an estimated software cost of $400,000. Considering the probing will be different, for each subsystem, the additional cost is estimated to be 12 subsystems × 15,000 per program or $180,000.

5. Software

a. Full System Emulator

The implementation of the Full System Emulator does not affect the software currently used in the AAH MP's. Therfore, the software discussions contained in Para. B-5 above applies to this design.

b. Automatic Program Translation

The use of the MIS requires translation or rewrite of all the existing software. The Proposed Automatic Program Translator is considered essential to this option. The Automatic Program Translator is discussed in Appendix B and the MIS in Appendix C. Design of the Automatic Translator is estimated at $450,000 plus $36,000 code generator for the Z-80 and $25,000 computer time. Updating the MIS to include all the current MP instruction sets would cost $50,000. Thus, this option (Alternative I) costs $561,000. The addition of a HLL optimizing compiler can be added when the language is selected. The compiler cost is estimated to be $500,000.

C. Maintenance

Software maintenance for the AAH covers two principal categories. These are complete rewrite due to mission and other changes and software changes required by ECP's. The data for the proposed system is extracted from Para. B-5 above.

ECP	
1st yr.	$4M -$7.5M
2nd yr.	1.7M - 3.4M
Total 8 yrs.	6M
ECP Sub Total	$11.7M - 16.9M
Rewrite Software (2x)	4.4M - 4.4M
Total 10 yr Life Cycle Cost	$16.1M - 21.3M

E. Packaging

The SOW requires a degree of repackaging as long as the overall configuration is not changed. Even though the Am 29116 is a single 52 pin DIP, a MPMP requires a number of supporting devices such as microcode memory, microcode sequencer and control logic. In addition to a physical PC board area limitation, the MPMP's require a considerable amount of power. Therefore, subsystems A, B, C, I, & N would require redesign of the power supply as well as the MP PC board. There is insufficient data to properly assess this problem area. Fortunately (or unfortunately), a technical meeting at US ARRADCOM, on 17 March 1980 provide information that subsystem K added a Z 80, 8 bit MP to the 2901A, 16 bit MPMP already in the subsystem. It thus appears that the Z 80 can be used in A, B, C, I & N with very little packaging problems. The use of an automatic circuit design aids such as the Algorex "AIDE" (9) will produce a new PC board layout plus extensive documentation at an estimated average cost of $30,000 per system. This results in an estimated repackaging cost of $480,000 for the 16 microprocessors.

F. Common Component Economics

The proposed common MP plus Z 80 design will provide other life cycle cost avoidance in area of component purchases. Based upon the new information disclosing the Z 80, the AAH now has 16 MP's and 12 assembly languages. It is assumed that all the present MP's in subsystems D, E, F, G, H, J, K, L, M, & O can be replaced by the Am 29116 and the MP's in subsystems A, B, C, I, N & K can be replaced by the Z 80. This then results in 11 Am 29116 MP's and 6 Z 80 MP's per system. Based upon 3000 systems plus 20% spares, the potential quantities to be purchased are 36,000 Am 29116's and 21,600 Z 80's. In addition, RAM, ROM and PROM memories will exceed 200,000 devices.* It is estimated that a 20%-25% reduction in parts cost would result. At an average of $50 per device, the current AAH Fire Control System for 3000 systems plus 20% spares are estimated to cost approximately 257,600 × $50 = $12.88 million. The proposed common hardware is estimated to cost $10.3 million.

G. Economic/Trade-off Analysis

The Economic/Trade-off Analysis is presented as a Matrix in Table I.

H. Cost Avoidance

1. First year cost avoidance thru full implementation of the proposed common MP, MIS, Automatic Translator, and optimizing HLL compiler is estimated between $3.5M and $7.5M with a potential additional cost avoidance for ATE software between $1.5M and $2M for a total of $5M to $9.5M.

2. Second Year cost avoidance thru full implementation as above is estimated between $1.7M and $3.4M.

3. Common MP hardware purchases should result in a cost avoidance of approximately $2.6M for production and initial spares provisioning.

---

*The component count is considered the same even though one 29116 will replace 4-2901A's. Thus the total MP related component count is in excess of 250,000 devices.

Areas of Comparison	Present Multiple Micro-processors, Independent Hardware and Software	Common Micro-Processor Full System Emulator Independent Software	Common Software Language Automatic Translator, Compiler Independent Hardware	Common Micro-Processors Common Soft-ware Automatic Translator Master Instruc-tion Set--Optimizing Compiler
Development				
Common Microprocessor	0	1.2	0	.8
Common Software	0	0	$1.40	.925 (Alt.I)
Software Maintenance	33.8 - 22.9	33.8 - 22.9	25.3 - 17.2	16.9 - 11.7
Software Development System	.52	.52	.03	.03
Program Rewrite	28.4	28.4	22.6	14.2
Software Documentation	.4	.4	.03	.03
TM's/FM's--Hardware	1.3 - .5	.08	1.3 - .5	.08
TM's/FM's--Software	1.9 - .64	1.9 - .64	.12	.12
ATE Software	2.5 - 1.9	.6	2.5 - 1.9	.6
Depot Training	.3 - .18	.04 - .03	.3 - .18	.04 - .03
Supply Line Items Cost	24 - 18	6	24 - 18	6
Total 10 Year Life Cycle Cost	93.1 - 73.4	62.9 - 60.8	77.6 - 62.0	39.7 - 35.0

Table I. Economic/Trade-Off Matrix (Cost in Millions of Dollars)

ASSOCIATES, INC.

SECTION VII

SUMMARY AND RECOMMENDATIONS

The economic/trade-off analysis was based upon the comparison of the life cycle costs of the current heterogeneous mix of seventeen microprocessors composed of ten different hardware configurations and twelve (12) different assembly languages proceeding thru production and into the field and (1) common hardware (one type MP) with the current twelve different assembly languages and (2) the current ten different hardware configurations with a common assembly language and Automatic Program Translator and (3) a proposed common hardware design (one type MP) with a common assembly language and Automatic Program Translator. The detailed technical design concepts for the common Hardware, Automatic Program Translator, common assembly language (Master Instruction Set) and Software Aids are presented in Appendices A thru D.

Table I, Section VI is a matrix showing the above comparison. The proposed common Hardware with a common assembly language and Automatic Program Translator will result in a cost avoidance in excess of $40 million over the 10 year life cycle of the AAH. Cost avoidance between $5 million and $9.5 million could be realized during the first two years after fielding the AAH.

It is therefore recommended that:

1. The term "Integral Processor" be adopted as an approved description for microprocessors integrated into the PC board packaging with other components. Provide appropriate means for specifying deliverable hardware and software documentation during development and production.

2. Initiate the development of the Automatic Program Translator with the Master Instruction Set immediately. This will result in significant software maintenance cost avoidance regardless of which configuration AAH Fire Control System is fielded.

3. Initiate the design of the proposed common MP implementing the Master Instruction Set, using the Am29116 and ALGOREX AIDE^{R}. This will permit demonstration of the power of the common MP design as

well as the ability to repackage typical sub-system MP's.

4. Establish a program and schedule to phase-in the common MP into each sub-system.

REFERENCES

1. Department of Defense Directive No. 5000.29

   Subject: Management of Computer Resources in Major Defense Systems, dated April 26, 1976.

2. Department of Defense Instruction No. 5000.31

   Subject: Interim List of DoD Approved High Order Programming Languages (HOL), Date November 24, 1976

3. DARCOM Regulation No. 70-16

   Subject: Management of Computer Resources in Battlefield Automated Systems, Department of the Army, Headquarters U.S. Army Material Development and Readiness Command, 16 July 1979.

4. Critical Item Development Specification for Fire Control Computer YAH-64, AMC-DC-AAH-H3003B Code Identification 02741, 31 October 1978.

5. Report No. 79-105 by R. J. Brachman Associates, Inc.

   Subject: Commonality Study of Computer Hardware Resources in the Advanced Attack Helicopter (YAH-64) Fire Control System, October 24, 1979

6. William J. Harmon, Jr. and Warren J. Miller, "A High Performance 16 bit Bipolar Microprocessor" the Am29116, American Micro Devices, Sunnyvale, California.

7. Build a Microcomputer, Chapters 1-9, Advanced Micro Devices, 1978.

8. Eugene R. Hnatek, Semiconductor Memory Update--Two Part Series ROMS/RAMS, Computer Design, December 1979, January 1980.

9. Algorex Corporation Brochure, Algorex Corporation, Syosset, New York, 1976.

10. Report No. 79-104 of R. J. Brachman Associates, "Commonality Study of Computer Software Resources in the Advanced Attack Helicopter (YAH-64) Fire Control System," October, 1979.

11. Jacques S. Gansler, Deputy Assistant Secretary of Defense (Material Acquisition) Symposium on Computer Software Engineering Dated April 20, 1976, Polytechnic Institute of New York, New York City, N.Y.

12. Max Schindler, High Level Languages Fuel Increasing Microcomputer Real Time Applications, Electronic Design, January, 1980.

13. Second Software Life Cycle Management Workshop sponsored by U.S. Army Computer Systems Command and the IEEE Computer Society, Atlanta, Georgia, August 21-22, 1978.

14. Max Schindler, "Focus on software: While Problems Abound, So Do Solutions--If You Can Find Them," Electronic Design, March 15, 1979.

15. Dr. Barry W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973.

16. Paul Oliver, "Examining Programming Costs," Computer Decisions, April, 1978.

17. Jan Snyders, "Slashing Software Maintenance Costs," Computer Decisions, July 1979.

18. Defense Focal Point, Department of the Army, Project Master Plan for Operational Performance Analysis Language System, Dated 28 February 1975.

19. Martin A. Goetz, "Advanced Commercial Applications in the 80's," Datamation, 1979.

20. R. J. Brachman, Memorandum FCF-7-76, The Economics of Introducing Automatic Test Support System (ATSS) Into the Army Maintenance System, Frankford Arsenal, Philadelphia, Pa., Dated 13 April 1976.

APPENDIX A:

REPORT OF A TASK STUDY ON FEASIBILITY OF DEVELOPING A COMMON MICROPROCESSOR (MICROPROGRAMMED EMULATOR) WITH COST ESTIMATES FOR THE U.S. ARMY ADVANCED ATTACK HELICOPTER FIRE CONTROL SYSTEM

INVESTIGATORS: DR. RICHARD E. MERWIN

MR. JAGAN SUD

This page was left blank intentionally.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

The purpose of this task group study is to demonstrate the technical feasibility of developing a common microprocessor to replace the various microprocessors (MP) used in the 14 subsystems of the Advanced Attack Helicopter (AAH) Fire Control System. It was determined that the current MP's could be emulated using microprogramming techniques. There are 14 separate subsystems composed of seven different hardware microprocessor configurations, which also result in ten different software MP configurations. (1,2) The government-furnished data provides sufficient software information for six MP's. The six MP's cover the range from the simplest to most complex MP requirements, therefore, the design developed in this task group study is considered valid as the common MP for all AAH Fire Control subsystems.

The Fire Control Computer instruction set and instruction execution times were specified in the government document entitled, "Critical Item Development Specification for Fire Control Computer," YAH-64, No. AMC-DC-AAH-H3003B, Date 31 October 1978. It is therefore determined that the Fire Control Computer and the back-up Fire Control Computer represent the most critical performance requirements. Thus, the principal design effort described in this study is directed toward equaling or improving the specified performance requirements. The simplest configuration of the common MP is described briefly just to demonstrate that the parts count and capability can be reduced as required by particular subsystems.

The emulator design is based on the new (not yet released) Advanced Micro Devices (AMD) Am29116 CPU device. This microprogram controlled device has a 16-bit wide data path, 32 general purpose registers, a barrel shifter, and a 16-bit arithmetic logic unit (ALU) with a wide range of arithmetic and logical operations.

In addition to the design of the emulator hardware which is based upon a selection of AMD chips, an estimate has been made of the number of micro-instructions required to interpret the machine instruction set of the six microprocessors being emulated. A cost estimate of generating the microprogrammed emulators and a discussion of the feasibility of using a high-level language (HLL) to microcode compiler to generate the emulators concludes the report.

The approach taken to design the emulators was to first define a hardware system that contained all the primitive operational functions required to represent each of the six microprocessors. Using the bit slice approach developed by AMD permits great flexibility in defining data paths, register sizes, and levels of hardware control. The next step was to define the micro-instruction control word format which supports the execution of the primitive functions contained in the microprocessors to be emulated. Each microprocessor internal architecture is mapped onto the proposed hardware system including ALU operations, memory management, register connections, interrupt processing, data paths, and shift and status bit manipulations.

The design of the microprogrammed interpreters for each of the six microprocessors is based upon the available internal operations of the Am29116 along with the other supporting chips. Such capabilities as interrupts, direct memory access, and microsequencing must be factored in at this stage to account for their interaction with the micro-instruction control functions. The basic control flow consists of first an instruction fetch cycle which utilizes an instruction address stored in a program status word (PSW) or a register referred to as a program counter (PC). The next operation is to determine the addressing mode used to define the address of the operand to be fetched from storage (if required). Finally, the required arithmetic, logical, or shift operation is performed to complete the machine instruction interpretation cycle.

The cost estimate for the generation of the six emulators is based upon the number of micro-instructions categorized in terms of difficulty of generation. In general, the cost of generating microprograms is much higher than the cost of conventional programming. Systems programming is generally regarded as the most difficult type of conventional programming and microprogramming is more complicated and will cost proportionately more. This topic will be dealt with below in more detail.

In view of the high cost of generating microprograms, the use of tools to reduce this cost is highly desirable. There have been some recent developments in generating microprograms directly from the PASCAL high-level programming language. The use of this tool to generate either emulators or microprocessor machine language will be described. A particularly interesting approach is to generate versions of the present software programs for the six microprocessors in terms of the "Master Instruction Set" (MIS) language (3) proposed by R. J. Brachman Associates, Inc. A simple one-time translation would then convert this standard representation into each microprocessor's machine language. Program maintenance could be either at the HLL or MIS level.

EMULATOR HARDWARE DESIGN

The design of an emulator capable of replacing six existing microprocessors must begin with the selection of hardware components. The six microprocessors being replaced are:

1. Motorola 6802
2. Intel 8080/85
3. TI SBP 9900
4. Am2901A - Back-up Fire Control Computer*
5. MECA 43 Fire Control Computer
6. 54LS 181 Special Chip

*There are four other microprocessors configured using the Am2901A four-bit slice device. These have equal or lesser capability than the back-up Fire Control Computer.

These units cover a wide spectrum of hardware design including eight and 16 bit data paths, up to 16 levels of interrupt capability, direct memory access, and up to 16 general purpose registers.

In order to replace this wide range of microprocessor capabilities the new (3rd/4th quarter 1980) Am29116 CPU chip was selected. This 52-pin device features up to 32 general purpose 16-bit registers, 16-bit data paths, arithmetic and logical operations, and a barrel shifter. An internal control line decoder supports a wide range of internal functions based upon 16 input control lines driven by an external control word storage unit.

In addition to the CPU, a wide range of other emulator support functions must be provided to meet the Fire Control Computer requirements. The Am2910 chip provides microprogram sequencing and branch control, control of the instruction and control word registers, and input to mapping PROMs for condition codes and device priority. Interrupt processing requires two Am2914 and one Am2913 device to accomodate up to 16 levels of interrupt priority while an Am2940 provides direct memory access (DMA) functions. The chips noted above provide control within the emulator and another group of chips provide such functions as instruction register, data bus interface, variable cycle system clock, and multiplexors providing data path control.

A hardware functional block diagram of the emulator design is shown in Figure 1. This shows the principal data flow paths along with the control lines. It is assumed in this design that the random access memory (RAM), read only memory (ROM), and the I/O system controls are accessed via a data bus and address bus and appear external to the emulator. A more detailed block diagram of the emulator is shown in Figure 2a and the I/O and memory management system is shown in Figure 2b. At this level all the data paths and control lines are clearly shown along with the AMD device numbers for all the hardware components. A list of AMD devices and their functions is shown in Table I. A minimal configuration would use the devices marked with an asterisk. The block diagram of this configuration is shown as Figure 2c.

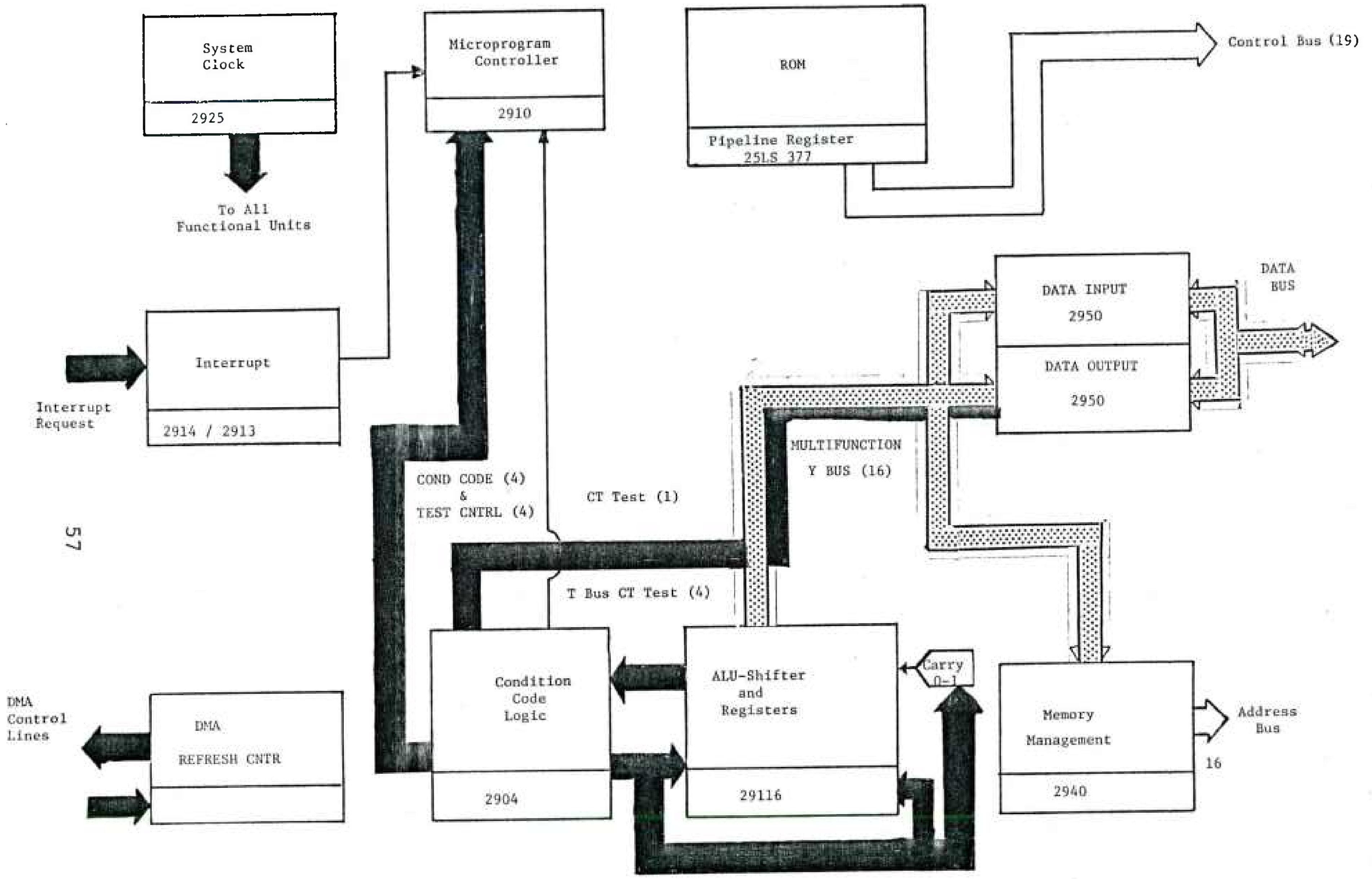


FIGURE 1: Emulator Hardware Functional Block Diagram

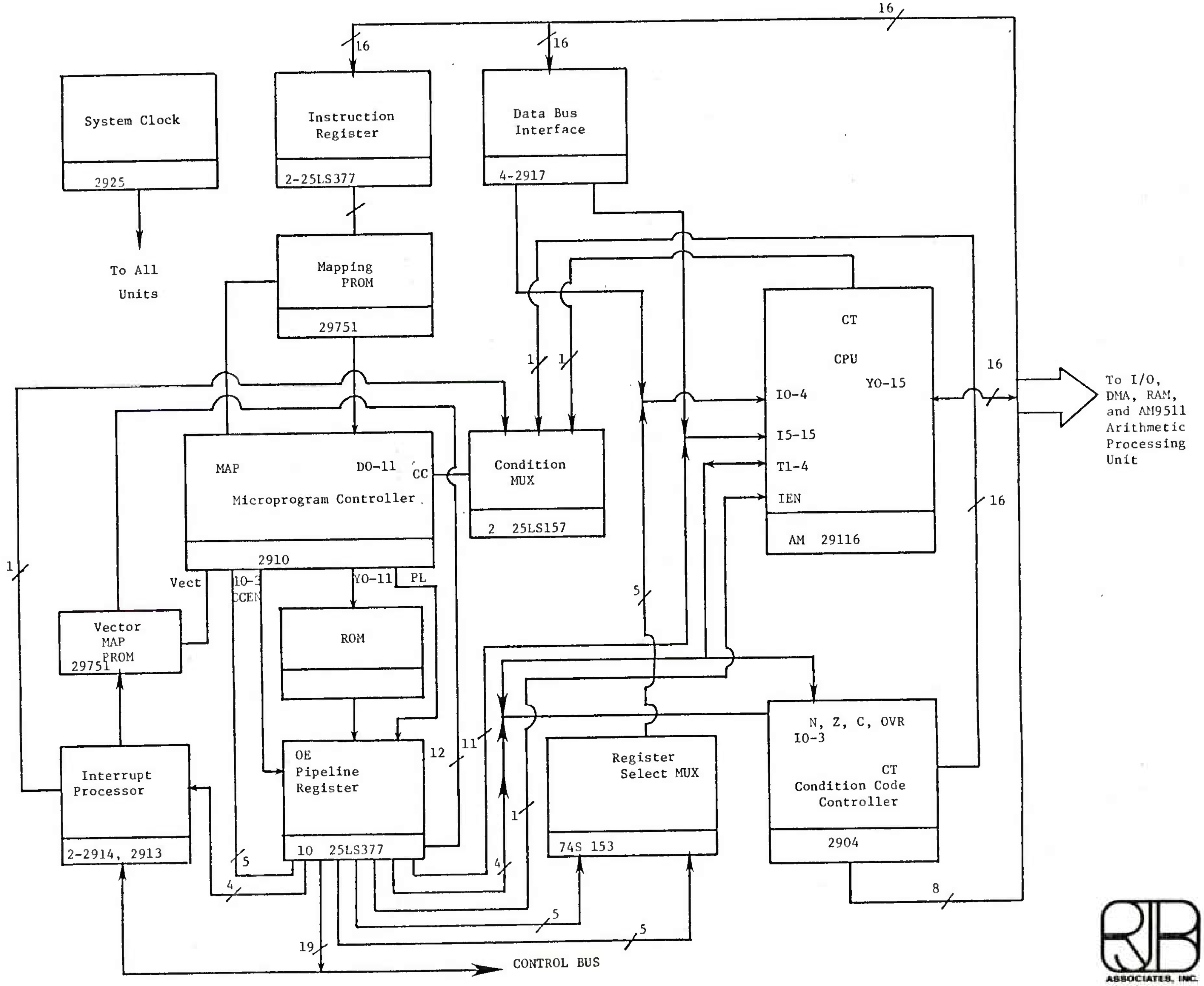
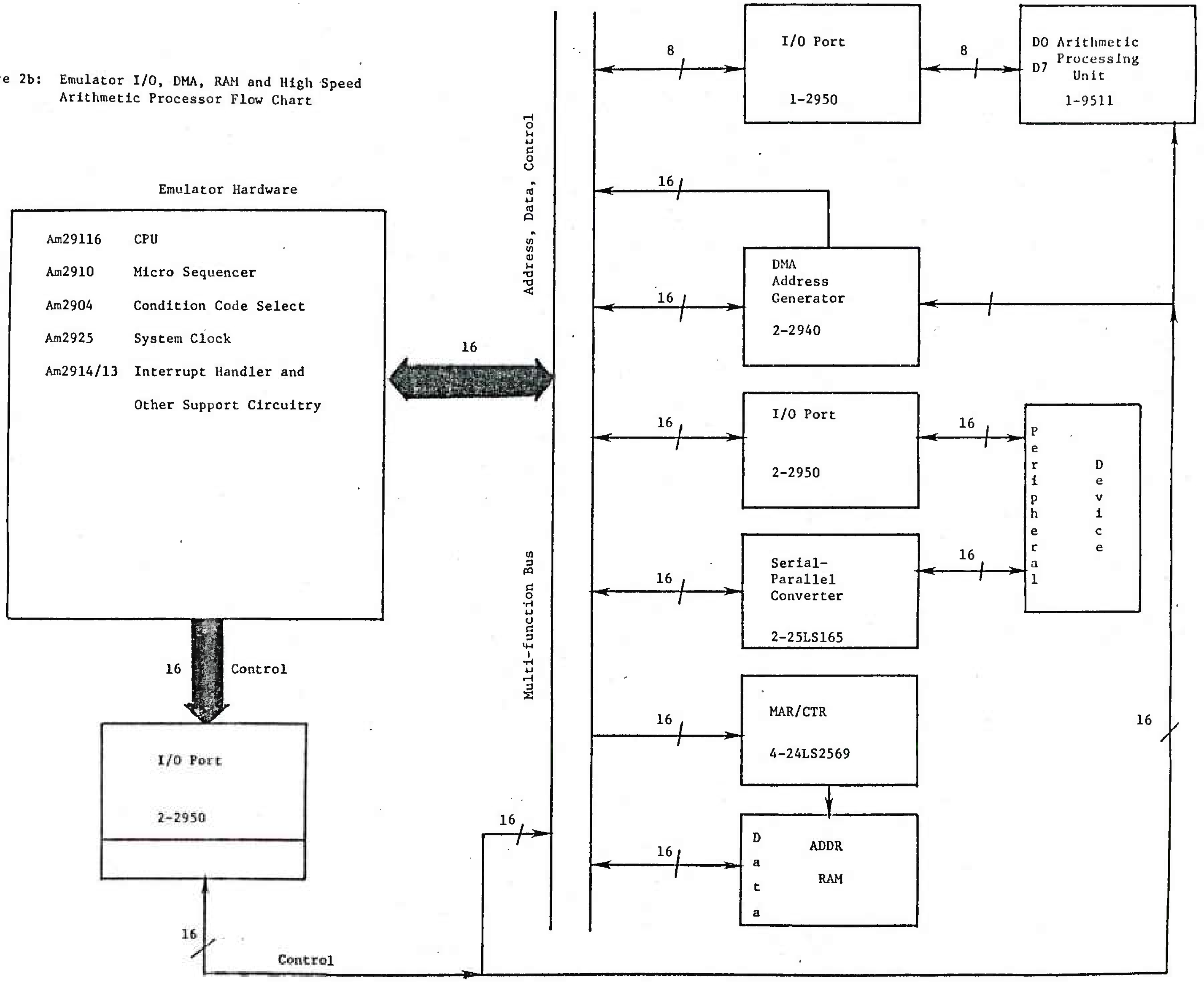


Figure 2a: Functional Control/Data Flow Diagram

Figure 2b: Emulator I/O, DMA, RAM and High Speed Arithmetic Processor Flow Chart



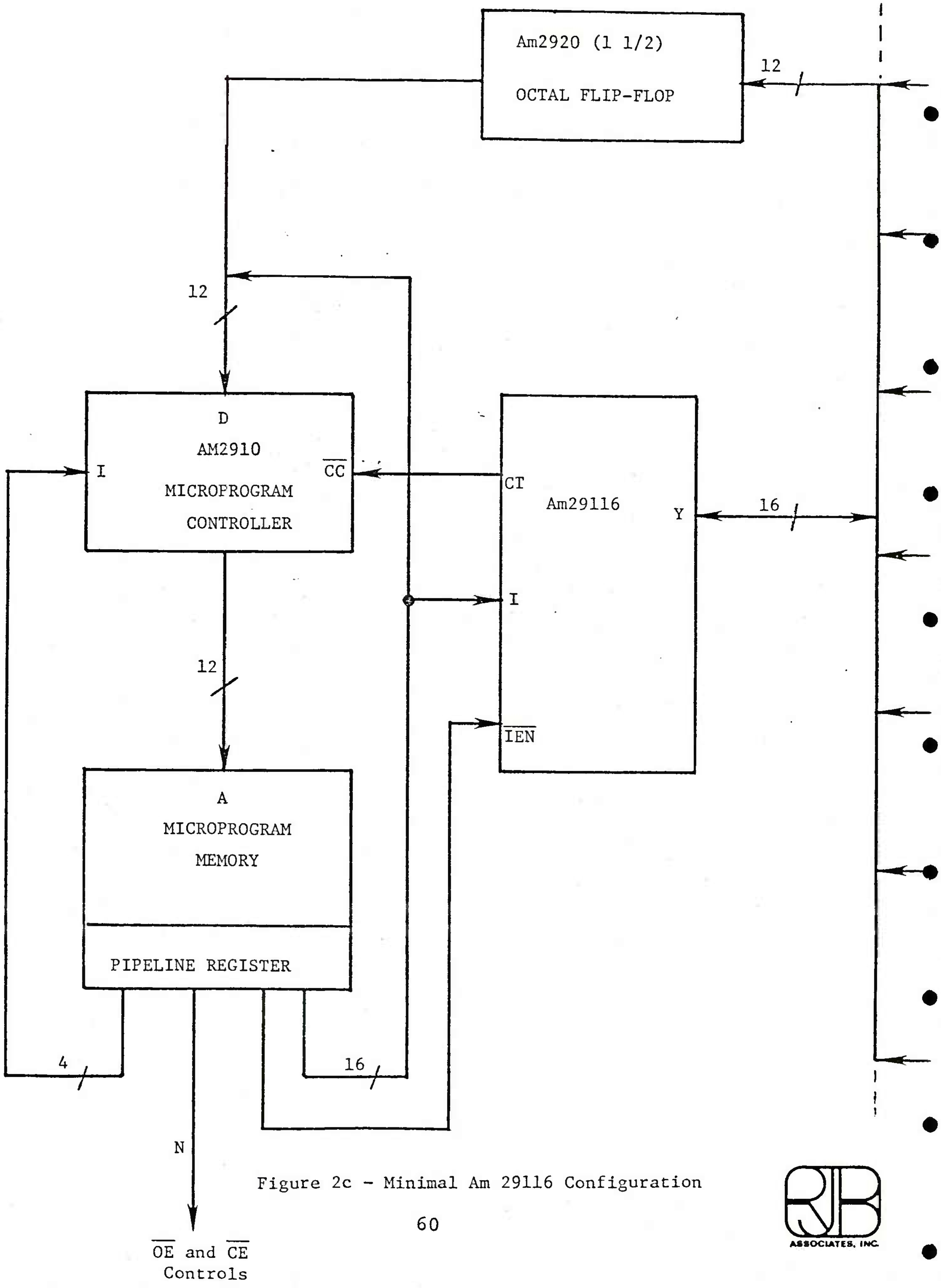


Figure 2c - Minimal Am 29116 Configuration

TABLE I

AMD DEVICES REQUIRED TO IMPLEMENT
MICROPROCESSOR EMULATOR

CHIP #	DESCRIPTION	QUANTITY	EMULATOR FUNCTION
Am 29116*	16 Bit Bipolar Microprocessor	1	CPU
2904	Status and Shift Control Unit	1	CPU Status Control
2910*	Microprogram Controller	1	Microprogram Controller
2913	Priority Interrupt Expander	1	Priority Interrupt Expander
2914	Vectored Priority Interrupt Encoder	2	Vectored Priority Interrupt Encoder
2917	Quad Three-State Bus Transceiver	4	Data Bus Interface
2925	System Clock Generator and Driver	1	System Clock Generator and Driver
2940	DMA Address Generator	2	DMA Address Generator
2950	8 Bit Bidirectional I/O Ports	5	I/O Data and Control Bus
29751	256 Bit Bipolar PROM	2	8X32 Vector Mapping PROM
25LS157*	Quad 2 Input Multiplexor	2	Register Address MUX
25LS164	8 Bit Serial-In, Parallel-Out Shifter	1	I/O Serial Parallel Converter
25LS2569	4 Bit Up/Down Counter	2	Memory Address Register and Control
25LS377	8 Bit Register	2	Instruction Register
74S153	Dual 4 Input Multiplexor	10	Microprogram Pipeline Register
		2	Condition Code MUX

Total of 37 Chips

A major concern in the emulator design is the exact capabilities of the Am29116 CPU device. Our information (4,5) spells out in considerable detail the various functions performed by this device along with its use in several representative applications. A block diagram of the device is shown in Figure 3 and from this, along with the applications examples, the functional assignment of the 52 pins as shown in Figure 4 was deduced. These assignments are intended simply to be representative and won't correspond to the actual assignments to be specified by AMD when the chip specifications are made available.

Some further discussion of the 16 Am29116 control word inputs is required. Five of these inputs are required to select one of the 32 general purpose registers. At least five control inputs are required to specify the ALU and barrel shifter functions. The balance of five to six pins is required to control the three muliplexors controlling the input to the ALU along with the carry, zero detect, status, test, and conditional test multiplexors. Another input is assumed to select the byte or word mode. This assumes that the Am29116 has an internal decoder which converts the signals on the four to five signal pins available into the required number of lines to control the multiplexors noted above. In summary, a number of assumptions had to be made regarding the internal operation of the Am29116. It is believed, however, that these assumptions are conservative and that the impact on the design of the emulator microprograms of misconceptions about the control of the Am29116 will be minimal. (This has been confirmed by AMD since the completion of this report.)

Another version of the emulator hardware was designed to incorporate the Am9511 floating point arithmetic device. This unit provides floating point and trigonometric calculations to be executed off line. This insures that the solution of the Fire Control Computer algorithms can be provided equal to or faster than the present design. Future detailed studies may show that the solutions do not require the separate Am9511 device. The data is supplied via the I/O system and the results are returned via the same path and will require control inputs from the microprocessor control word.

The first phase in the design of the microprocessor emulator was to specify the bit assignments for the micro control word. Most of the chips used in the hardware implementation require inputs from the micro control word. As can be seen from Figure 4, the Am29116 requires 25 control lines; however, if the register-to-register transfer capability is to be implemented, then five more control bits must be added to the micro control word, bringing the total number of control bits assigned to the Am29116 up to 30.

Figure 5 (a and b) shows the format of the micro-instruction control word, which indicates that 80 bits are required to control the seven major chip types along with register, multiplexor, and storage read write controls, and the optional arithmetic unit. It is possible that some economies in the number of control word bits required could be achieved by use of decoders to drive the individual control line assignments. Because of restrictions on the number of chips that can be accomodated on the printed circuit board containing the emulator, it was decided to not seek economy in micro-instruction bit counts in favor of reducing the number of chips required to implement the emulator.

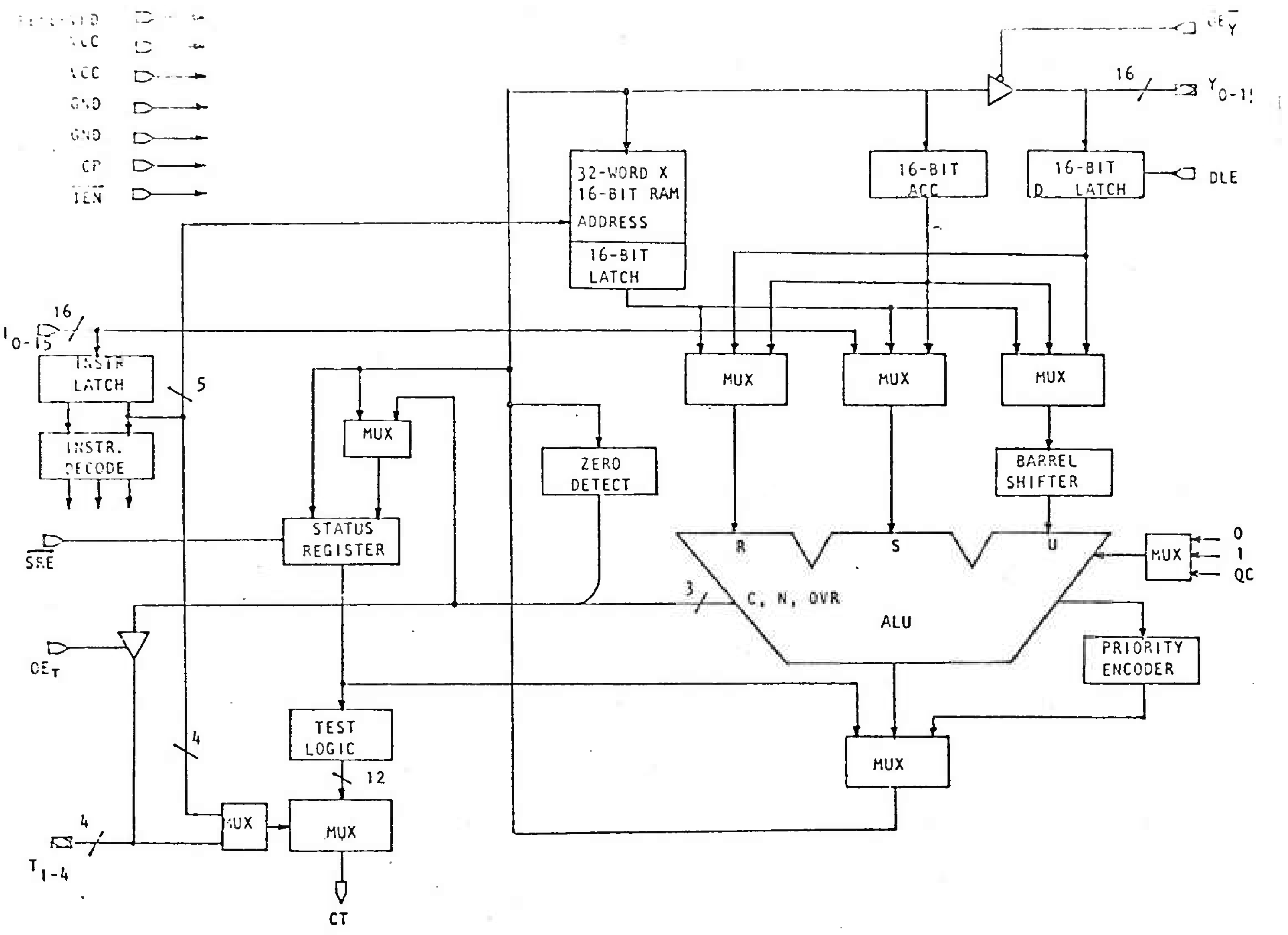


FIGURE 3: Am29116 Chip Architecture

FIGURE 4: Assumed Am29116 Pin Assignments

		Pin #	Pin #		
Reserved		1	52		Not Defined at Present
	VCC	2	51		
	VCC	3	50		
	GND	4	49		
	GND	5	48	Y0	Data/IO
Clock Pulse	CP	6	47	Y1	
Status Register Enable	SRE	7	46	Y2	
Output Enable	OET	8	45	Y3	
	I15	9	44	Y4	
	I14	10	43	Y5	
	I13	11	42	Y6	
	I12	12	41	Y7	
	I11	13	40	Y8	
	I10	14	39	Y9	
	I9	15	38	Y10	
	I8	16	37	Y11	
	I7	17	36	Y12	
	I6	18	35	Y13	
	I5	19	34	Y14	
	I4	20	33	Y15	
	I3	21	32	T1	Test Enable
	I2	22	31	T2	
	I1	23	30	T3	
	I0	24	29	T4	
Data	OLE	25	28	CT	Cond. Code Output
Output Y Bus Enable	OEY	26	27	IEN	Input Enable

ASSOCIATES, INC.

Bit #	Chip #	Pin Name	Function
0		D0	
1			
2			
3			Shared Control Lines
4			
5			
6			Control Memory ADDR
7			I/O Branch ADDR
8			
9			
10			
11	Am2910	D11	
12		I0	
13			
14			2910 Control Input
15		I3	
16		PL	Pipeline ADDR Enable
17		VECT	Int. VECT Promenable
18		CCEN	Cond. Code Enable
19		MAP	Mapping PROM CTRL
20		CI	Carry In
21		RLD	Register Load
22		OE	Output Enable
23		EZ	
24		EN	
25		EC	Condition Codes
26		EOVR	
27		CEN	Enable Micro Status Register
28		CEM	Enable Machine Status Register
29	Am2904	I0	
30			
31			2904 Control Input
32		I3	
33			IR Register Control
34	Data Path		Data Bus Transceiver Control
35			Cond. Code MUX
36	Control		ALternate Register Addr. MUX
37	Lines		Mem. Write
38			Mem. Read
39			Fetch Cycle

FIGURE 5 a: Micro-instruction Word Layout

Bit #	Chip #	Pin Name	Function
40		I0	Register Select Lines
44		I4	
45			Control Inputs
49			
50	29116		
54			
55		I15	
56		I'0	Alternate Register Select Lines
57		I'1	
58		I'2	
59		I'3	
60		I'4	
61		T1	Test Control Lines
62		T2	
63		T3	
64		T4	
65		IEN	INSTR Input Enable
66		OEY	Y Bus Output Enable
67		DLE	Data Latch Enable
68		OET	Output Test Enable
69		SRE	Status Register Enable
70		I0	Interrupt Control Lines
71			
72	2914		
73		I3	
74		IE	Instruction Enable
75		I0	DMA Control Lines
76	2940		
77		I2	
78		OEA	Output Enable
79			

FIGURE 5 b: Micro-instruction Word Layout

MICROPROCESSOR ARCHITECTURE MAPPING

The next phase in the design of an emulator for the six microprocessors consists of mapping these architectures onto the emulator hardware. Before describing this procedure a slight digression is in order to introduce the concept of the target and host machine as described by Davies (6). The emulators for the six microprocessors are implemented on the host machine described in the previous section. The target machines are the six microprocessors. The machine language instruction set for each microprocessor along with addressing modes are interpreted on the host machine, i.e. Am29116, through microprogramming. This can be accomplished through the generation of microprogrammed interpreters for each of the six target microprocessor machine instruction sets. These may be simultaneously resident in the control storage of the host machine or can be individually resident. Figure 6 pictorially illustrates the target-host machine relationship.

For the hardware architectures of each target machine to be easily emulated on the host machine requires that certain hardware features be available in the host machine. The data paths, register sizes and number, internal data formats, addressing modes, ALU functional operations, and handling of flag and status conditions must be implementable within the host machine hardware for all the target machines. Other considerations include execution times for the ALU and storage units, register transfer rates, and other functions which take place in the micro-instruction cycle.

Fortunately the Am29116 CPU meets a wide range of hardware requirements and features a 100 nanosecond micro-instruction cycle along with 32 general purpose registers and 16-bit data paths. The associated AMD bit slice LSI components provide great flexibility in meeting a wide range of required host machine architectures.

The internal architecture, addressing modes, interrupt levels, instruction repertoire, and I/O characteristics of the six microprocessors to be emulated are shown in Table II. To map the target machine architecture to the host machine, it is necessary to assign the registers of the target machine to the registers of the host machine. In carrying out this procedure an attempt was made to assign common register function to one register in the Am29116. Specifically, the program counter, stack pointer, index, accumulator, and status register assignments were made to specific Am29116 registers. Other register assignments were arbitrary. Since the Am29116 has 32 general purpose registers, no difficulty was encountered in making the assignments as shown in Table III.

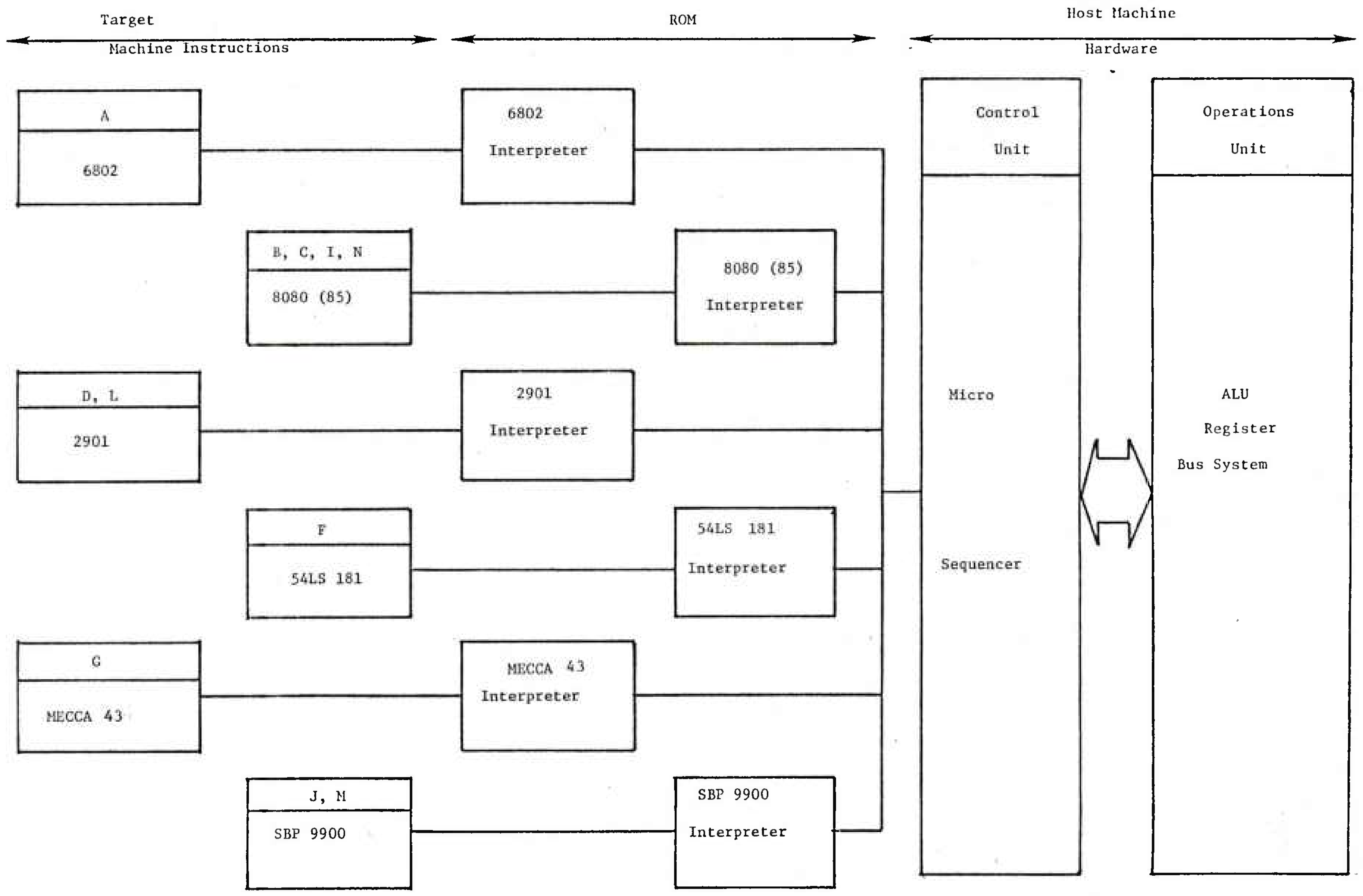


Figure 6: Host-Target Machine Relationship

TABLE II Target Machine Characteristics

Alphabetic Classification	A	B, C, I, N	D, L	F	G	J, M
Chip Type	6802	8080/85	Am2901A (SDP173)	54LS181	MECCA 43	TI SBP 9900
Configuration						
Registers - General Purpose	---	7-8 Bit	16-16 Bit		16-8 Bit	16-16 Bit
Index	1-16 Bit	---	1-16 Bit		3-16 Bit	R0-16 Bit
Accumulator	A and B 8 Bit	1-8 Bit	1-16 Bit		1-8 Bit	1-16 Bit
Status	1-8 Bit	1-8 Bit		1-4 Bit	1-8 Bit	R15-16 Bit
Stack Ptr	1-16 Bit	1-16 Bit	1-16 Bit		1-8 Bit	
Prog Ctr	1-16 Bit	1-16 Bit			1-16 Bit	R14-16 Bit
PSW	---	1-8 Bit				
Buss Structure						
Data	8 Bit	8 Bit	16 Bit	4 Bit	8 Bit	16 Bit
Address	16 Bit	16 Bit	16 Bit		16 Bit	15 Bit
Control	8 Bit	8 Bit	8 Bit		11 Bit	16 Bit
Instructions	72	80-8085 78-8080	101	32	88	72
Addressing Mode				Not Applicable		
Reg to Reg	Yes	Yes	Yes		Yes	Yes
Direct	Yes	Yes	Yes		Yes	Yes
Indirect	Yes	Yes	Yes		Yes	
Indexed	Yes		Yes		Yes	Yes
Immediate	Yes	Yes	Yes		Yes	
Extended	Yes					
Implied	Yes	Yes				
Relative	Yes					
Auto Increment						Yes
Auto Decrement						Yes
I/O	Memory Mapped	Direct	Memory Mapped	Direct	Direct	Separate Bus
Interrupt Levels	3	8	16	2	16	16
Type	Polled Vector'Fetch	Vectored Only	Direct	Direct	Vectored	Priority Vector

TABLE III   Am29116 To Target Machine Register Mapping

Host	A	B, C, I, N	D, L	F	G	I,M
Am29116	6802	8080/85	Am2901A (SDP173)	54LS181	MEGCA 43	TI SBP 9900
Register #						
1		W	GP1		GPA1	Work Space Reg 1
2		Z	GP2		GPA2	Work Space Reg 2
3		D	GP3		Base Ret A	" 3
4		C	GP4		B Register A	" 4
5		H	GP5		INDEX X	" 5
6		L	GP6		INDEX Y	" 6
7		ADDR LATCH	GP7		INDEX Z	" 7
8		TEMP	GP8	Not Applicable		" 8
9			CP9	System doesn't	GPB1	" 9
10			CP10	employ General	GPB2	" 10
11			CP11	Purpose	Base Ret B	" 11
12			GP12	Register	B Register B	" 12
13			CP13		INDEX X	" 13
14			GP14		INDEX Y	" 14
15			CP15		INDEX Z	" 15
16			GP16			Work Space Reg 16
17	PROG CTR	PROG CTR			PROG CTR 1	PROG CTR
18					PROG CTR 2	
19						Work Space PTR
20	STACK PTR	STACK PTR				
21	IR EXT	IR EXT	IR EXT		IR EXT	
22	IR EXT	IR EXT	IR EXT		IR EXT	
23	IR EXT	IR EXT	IR EXT		IR EXT	
24	IR EXT	IR EXT	IR EXT		IR EXT	
25						
26						
27	INDEX					
28						
29						
30	ACC A				ACC A	
31	ACG B				ACC B	
32	STATUS REG	STATUS REG	STATUS REG		STATUS REG	STATUS REG

ESTIMATE OF EMULATOR MICRO-INSTRUCTION REQUIREMENTS

The design of emulators can be broken down into three major phases corresponding to the machine language instruction processing procedure. The procedures followed for each target machine instruction are collectively referred to as an interpreter which executes on the host machine and makes it look like the target machine to the user. Each machine instruction interpretation cycle consists of an instruction fetch phase, an operand fetch cycle, if required, and an execution phase.

The instruction fetch phase simply issues the address of the next machine language instruction to be executed to the memory address register and requests a memory read cycle. Along with this operation the instruction control counter is incremented by some amount proportional to the length of the machine language instruction that has been fetched. For variable length instruction formats this may involve decoding the instruction OPCODE to determine if additional segments of the machine language instructions must be accessed from memory. The instruction fetch phase is completed by entering the machine language instruction into the instruction register (IR).

The data operand access phase is only necessary for those machine language instructions calling for a memory reference. The entry into this phase is determined by the fetched instruction OPCODE. For most OPCODEs specifying register-to-register, stack, branch, test, and shift operations, no memory operand fetch operation is required. The memory access phase is often referred to as the addressability mode and a wide variety of possibilities exist as shown in Table II. Direct addressability simply uses an operand address specified in the fetched machine language instruction to access a data word. Indirect addressing implies that the operand being accessed is an operand address and this may be repeated many times. Indexed addressing involves adding a constant contained in a register to the address specified in the machine language instruction and a relative address is simply an offset of constant value added to the machine language instruction address operand.

Registers are often referred to as containing data operand addresses and these are used along with increment and decrement operations to cycle through sequences of data intermixed with instructions. Each target machine emulator must provide an implementation of the addressability modes of the target machine. If these involve many possiblities the corresponding emulator micro-instructicn count can be substantial.

The final phase of interpreting a machine language instruction is the execution of the functions specified by the OPCODE. It is assumed at this point that all operand data required is stored in registers and some arithmetic or logical function is to be executed on this data. The result of this execution is stored in a specified destination register or an accumulator or stack as dictated by the machine internal architecture. Typically for simple binary infix operators, e.g. ADD, SUBT, AND, OR, XOR, this operational phase requires from three to four micro-instruction systems. Thus for a target machine with fewer than 100 different machine language instructions the number of micro-instructions to implement an interpreter lies in the 150 to 300 range. Adding the micro-instructions required to implement the data operand addressability and instruction fetch functions brings the total micro-instruction count and corresponding PROM capacity requirements to the 300 to 400 80-bit control words.

The overall emulator operational flow is shown in Figure 7. As can be seen the instruction fetch and operand access cycle is common to each machine language instruction. The selection of the approporiate micro-instruction sequence corresponding to the 8-bit OPCODE is accomplished via an 8X32 mapping PROM (see Figure 2a). The output of the PROM gives the ROM address of the micro-instruction sequence. Each of these sequences is terminated with a branch micro-operation back to the beginning of the instruction fetch phase.

The specific estimate of number of micro-instructions required for the six emulators is shown in Table IV. The approach in making these estimates was conservative to allow for the many assumptions that were made in this design effort. The Am29116 CPU chip has many powerful internal features including the capability of executing internal register-to-register operations in one micro-instruction cycle (100 nanoseconds). While this feature was exploited in the emulator design, a more conservative approach was taken on other functional procedures.

A major problem with the design of the emulators is the lack of knowledge of the exact internal operation of the Am29116. This is reflected frequently in assuming two micro-instruction cycles instead of one. For example, an added cycle is assumed for shift operations although these are very likely executable in conjunction with ALU operations.

To illustrate the emulator design in more detail, detailed micro-instruction sequences for the Intel 8080(85) and Motorola 6802 are shown in Figures 8 and 9, respectively for the fetch, addressability mode, and a sample set of functional operations.

Transition

1. TURN ON
2. RUN MODE
3. INSTR FETCH
4. EXTENDED INSTR FETCH
5. INSTR TYPE DECODE
6. MEMORY ACCESS
7. EXECUTE INSTR
8. ENABLE INTERRUPT

Instruction Type

RR:	REGISTER TO REGISTER
RX:	RECISTER TO STORATE
RI:	IMMEDIATE
SF:	SPECIAL FUNCTION

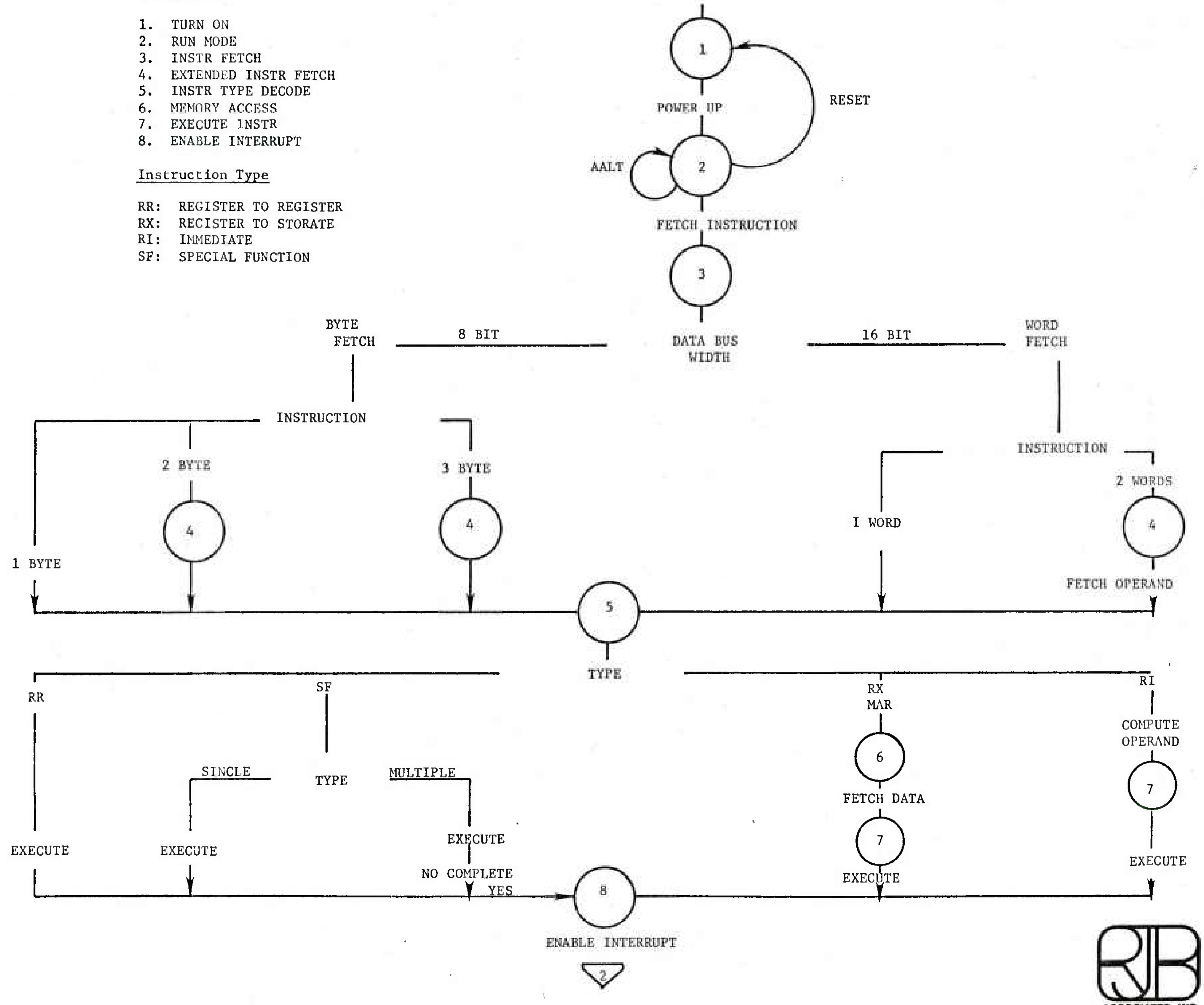


Figure 7: Emulator Flow Chart

TABLE IV: EMULATOR MICROINSTRUCTION COUNT ESTIMATES

Alphabetic Classification	A		B,C,I,N		D,L		F		G		J,M	
CHIP TYPE	6802		8080(85)		Am2901A (SDP173)		54LS181		MECCA 43		TI SBP 9900	
	INSTRUCTION		INSTRUCTION		INSTRUCTION		INSTRUCTION		INSTRUCTION		INSTRUCTION	
MACHINE INSTR CATEGORY	MACHINE	MICRO	MACHINE	MICRO	MACHINE	MICRO	MACHINE	MICRO	MACHINE	MICRO	MACHINE	MICRO
Arithmetic	16	40	15	69	25	38	5	25*	33	140	19	40
Logical	11	30	12	40	23	60	2	10*	10	28	8	15
Data Transfer	17	40	15(17)	86(92)	18	35	20	100*	27	60	19	30
Branch	16	19	10	16	15	20	12	60*	9	15	15	30
Skip	0	0	0	0	0	0			3	9	0	0
Return	2	10	9	33	2	6			0	0	0	0
Subroutine Call	2	16	10	29	2	10			0	0	1	2
Miscellaneous	8	30	7	30	11	44			6	28	7	40
Fetch Addressability		120		50		86				102		100
Total	72	305	78(80)	353(359)	96	299	39	195	88	382	69	257
MICRO INSTR/ MACH INSTR	4.24		4.53 (4.49)		3.11		5.0		4.34		3.06	

*Because of lack of information about instruction set for this system a factor of 5 micro-instructions per machine instruction is used here.

Figure 8: Motorola 6802 Microinstruction Flow Chart

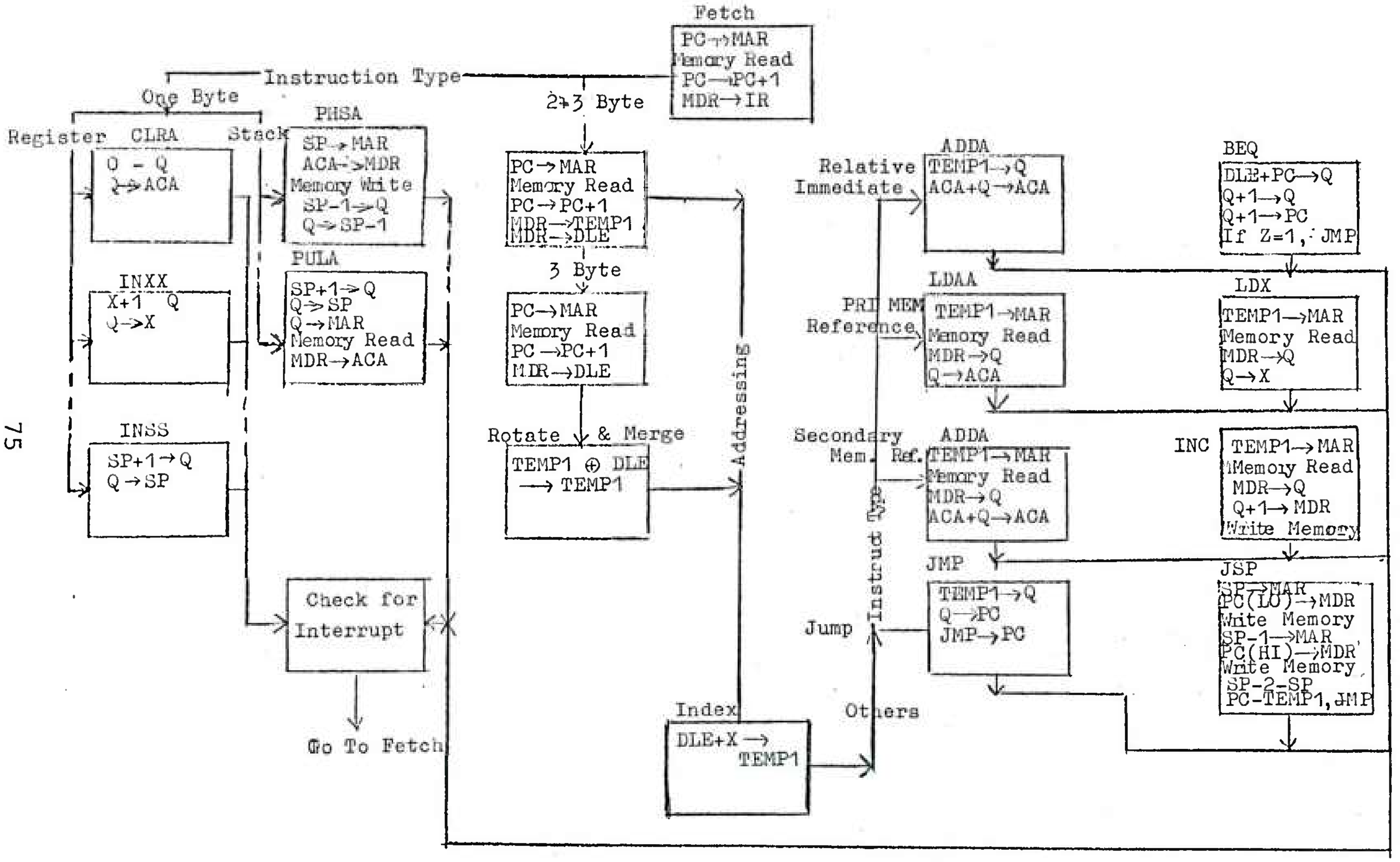
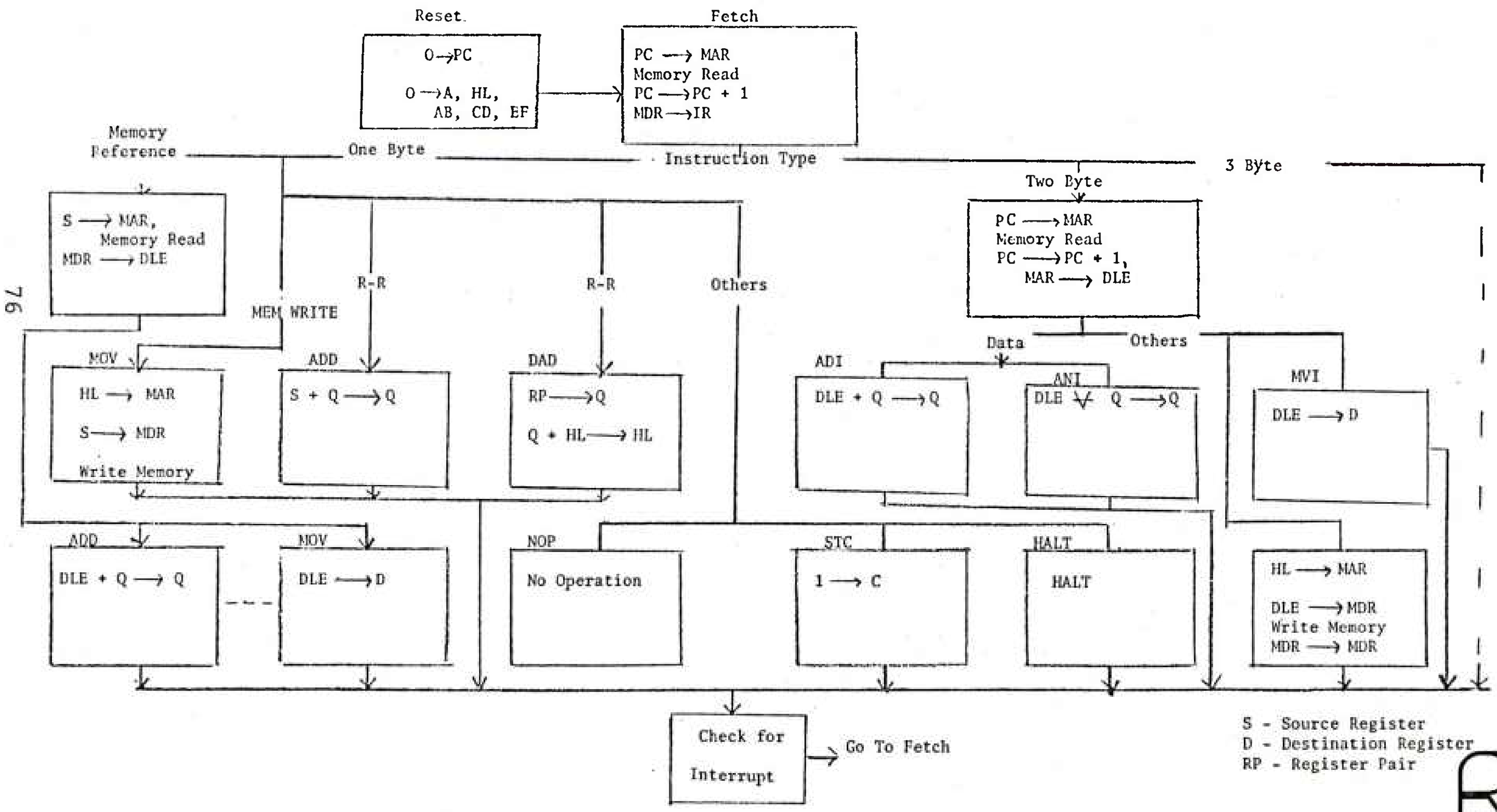


Figure 9: 8080(A)/8085(A) Microinstruction Flow Chart



S - Source Register
D - Destination Register
RP - Register Pair

EMULATOR DESIGN COST ESTIMATES

Accurate cost estimation procedures have long been a major obstacle to management of software development projects. The uncertainties associated with conventional software cost estimation are at least as prevalent in the estimation of microprogramming costs. Writing of microprograms is generally considered one of the most tedious and error-prone procedures in the development of computer systems. There is little or no data available in making such a cost estimate and computer manufacturers, who might have such data, regard it as highly proprietary.

The closest analogy to conventional software generation for use as a measure of the cost of microprogramming would be systems programming. This category of software generation is generally considered from three to five times more difficult than conventional programming. Further, it is frequently required that assembly language rather than a high-level programming language be used to carry out systems programming which further complicates the task. System programming production rates (7,8) have been estimated to be one-third instruction per programmer hour. This includes design, coding, implementation, validation and documentation. In this estimate, microprogrammer productivity will be assumed to range from one-third to one-sixth micro-instruction per hour. The former rate will be associated with portions of the emulator that are straightforward to generate. The latter rate will be assumed for the more difficult portions of the emulator. Three catagories of difficulty are defined: easy, medium and hard. Based on an hourly cost of a microprogrammer of $33.00/hr., this gives a cost of each micro-instruction as: easy -- $100.00, medium -- $150.00, and hard -- $200.00. Each emulator is broken down into nine categories and a difficulty assignment is made to each category. In Table V, the cost estimate for each emulator is shown along with a breakdown of costs by each of the nine categories as shown in Table V.

In addition to the generation of the microprograms for each emulator, it is also necessary to consider the support tools which will be required. These include a micro-assembler program, a micro-simulator program, and the documentation support system. Such systems are relatively straightforward to design and it will be assumed that they are written in a high-level programming language and execute on commercially available computers, e.g. IBM 370. A cost estimate for both a micro-assembler and simulator is included in the cost estimates shown in Table V. This leads to a cost of generating all emulators including support tools of $350,000.

If the MIS language were to be used as the only target machine language, as proposed elsewhere in this study, a single but more complex interpreter would be required. Referring to Figure 6, the six target machine instruction sets shown on the left would coalesce into the MIS language and the six interpreters shown in the middle could be replaced by one. This would reduce the size of ROM required to store the interpreters by as much as 70 to 80 percent and require less space on the ROM chip carrier.

The feasibility of designing a single microprogrammed interpreter to execute on the host machine shown in Figure 6 for the MIS language wasn't part of our study effort. Only preliminary cost estimates could be made and these indicate that this interpreter would be more complex than any of the six interpreters covered in this study. A conservative estimate would be from $60,000 to $80,000 and with the support software would lead to a total design cost for the

TABLE V: Emulator Cost Estimates

Functional Type	Skill Level	Number of Microinstructions	Cost
Arithmetic	E	352	$ 35,200
Logical	E	183	18,300
Data Trfr I/O	M	357	53,500
Branch	E	160	16,000
Skip	H	9	1,800
Return	M	49	7,350
Subroutine Call	H	57	11,400
Miscellaneous	H	172	34,400
Addressability	M	458	68,700
	Total	1797	$246,650
		Micro Assembler	50,000
		Micro Simulator	50,000
		Total Cost	$346,650

Skill Level Manpower Costs/Microinstructions

E	$100.00
M	$150.00
H	$200.00

Machine		Number of Microinstructions	Emulator Cost
A	6802	305	$ 43,600
B,C,I,N	8080(85)	359	50,500
D,L	2901A	299	41,650
F	54LS 181	195	24,500
G	MECCA 43	382	50,000
J,M	SBP 9900	257	36,400
		1797	$246,650

emulator as less than $200,000. The principle concern here is the requirement to provide interpretation capability for a set of special operation codes required to adapt the MIS language to each of the MP systems.

A final comment pertains to the design process of emulators. The first phase deals with the design of the emulator. Part of this activity, in a preliminary way, has been carried out in the preparation of this report.

The next step is to actually generate the microcode required for each emulator. This assumes the prior availability of a micro-assembler and some means of recording design levels, changes thereto, and release of final designs to a manufacturing organization.

As portions of the emulator are generated, a sequence of more encompassing simulations are required to verify the performance of the emulator and provide a means to detect and correct errors in design.

Finally, the installation and check-out of the emulator provides the verification of operational performance. This requires the replacement of the target machine by the host machine and its emulator and the software generated for the target machine is used to exercise the emulator system. These are complicated procedures and require highly qualified personnel to carry out the necessary verification steps. The cost estimates for generating each micro-instruction required for each emulator are assumed to contain allowances for all these steps in the design, generation, and installation of the emulator. To be conservative, it would be wise to add a 25 percent contingency factor to account for the complexity of generating microprograms and the complicated procedures required to verify their performance as replacements for the target machine.

One of the leading contributors to the cost of emulators is the tedious procedure required to generate individual micro-instructions. A tool in common use in generally conventional software is the high-level programming language. The increased number of machine language instructions required to represent a given algorithm as opposed to the same algorithm expressed in assembly language for a given target machine is felt to be a small sacrifice to achieve greatly improved programmer productivity. The same argument could hold for generating microprograms and this possibility is the topic of the next section of this report.

USE OF HIGH LEVEL PROGRAMMING LANGUAGE TO GENERATE EMULATORS

Through a series of research contract efforts (9, 10), it has been demonstrated that a high level language (HLL), such as PASCAL, can be compiled into a non-machine dependent intermediate language format and subsequently translated into either machine language to execute on a target machine or microcode to emulate the target machine on a universal host machine. Both of these capabilities have applications to simplifying the software and hardware structure of the set of microprocessors currently installed in the AAH Fire Control System.

By use of modern compiler development techniques referred to as translator writing systems (TWS), it is relatively straightforward and inexpensive to develop prototype compilers for a wide range of HLL input languages and to produce intermediate or machine language representations of algorithms expressed in the HLL. Several compilers have been developed which produce an intermediate instruction stream representation referred to as a QUADRUPLE. This machine independent instruction format consists of an operation part, two source operand addresses, and a result operand address. It is a relatively straightforward procedure to translate this format into various machine language representations or directly into microcode. The latter representation must be loaded into a special writable micro-control word storage and accessed through procedures within the host machine's control circuitry. Many modern day minicomputers, e.g. DEC VAX 11/780, HP 1000, Interdata 8/32, support the "user microprogram" concept (11, 12) and provide extra writable control storage, micro-assemblers, and simulators to assist in developing microcode. In spite of the fact that microprogramming critical software kernals, i.e. program segments executed frequently, can improve system performance by factors of up to 10, "user microprogramming" facilities aren't used very often. The primary obstacle is the difficulty encountered in writing and debugging microprograms.

With the advent of the HLL compilers that can generate microprograms as output, much of the manual difficulty of generating microprograms is eliminated. This escape from the tedium of writing microprograms is not accomplished without some loss in the efficiency of microcode produced by a compiler as opposed to hand generated microcode. To date, a detailed study of HLL to microcode compiler performance hasn't been tested against hand generated microcode. Research exploring this question is currently underway.

A further question arises as to the utility of using an HLL microcode complier to generate emulators. A major question concerns the ability of an HLL to concisely express the type of actions described by an interpreter at the register transfer level. It is quite possible that a special HLL may be required and this issue certainly commands some in-depth investigation as a fundamental research issue in emulator design.

To further consider how the techniques described above could be applied to the activity described in this report, two approaches are suggested. The first would be to use the HLL to generate a machine independent representation of the algorithms required by the various weapons and avionics control systems. Either the QUADRUPLE as noted above or the MIS language would be candidate machine independent representations. In either case this intermediate representation would then be translated directly into microcoded interpreters to run on the host emulator hardware.

The second approach would be to generate the target machine language programs from the machine independent representation of the operational algorithm. This would preserve the existing microprocessor hardware. Again a translator for each target machine would be necessary and would still require maintenance of each target machine language although it would eliminate having to maintain programmers who were competent to program in these languages.

The impact of the HLL compiler alternative on the cost of generating emulators is difficult to assess. The cost impact comes in two areas. The first is a reduction in the cost of generating the emulators while the second, and perhaps the most important, is the impact on the maintenance cost of the emulators. These maintenance costs weren't addressed in this study. In terms of the initial cost estimates of generating the six emulators, which was estimated above to be $350,000.00, the use of a HLL compiler would probably not reduce this figure by much. This is because the costs of developing the HLL compiler and evaluating its performance have large R&D components. Since these costs are hard to estimate, the overall costs of emulator development are not easily derived. Again, as noted above, the long-range impact of this design alternative could be very significant. Further research and life cycle cost studies must be carried out to demonstrate this conclusion.

In comparing the two approaches, several issues must be considered. Using a single HLL to express all the weapons system functions would be extremely desirable especially in the software maintenance phase because programmers would only have to be skilled in one language instead of six or so much more complex machine languages. The output of the compiler, whether target machine languages or microcode for a universal emulator, would only have to be understood by a few "expert" programmers who would deal with compiler "bugs" and target machine problems. The choice of an intermediate format, i.e. QUADRUPLES or MIS, must be carefully explored. Again, only one language definition is required to be documented and maintained and the main issue would be efficiency of translation of this intermediate format into the host machine microcode or machine language. As presently constituted, the support of software and hardware for the AAH Fire Control System is going to require many parallel activities involving documentation and maintenance of skills in several machine language and hardware systems. Replacement of these multiple software-hardware maintenance systems by one universal emulator and HLL support software system seems far preferable both in terms of life cycle costs and training and retention of the necessary skilled personnel. Clearly a one-time cost of switching to this new approach would have to be written off but the longer-term economic and personnel requirements appear to more than compensate for the short-term conversion costs.

CONCLUSIONS AND SUMMARY

This section will present a number of conclusions derived from the emulator feasibility and cost estimate study along with some comments and suggestions for future investigations. The feasibility study was of short duration and there was a serious lack of information on one of the target machines, 54LS 181, as is evidenced by the gap in Tables II, III, IV, and V. It is believed, however, that the central theme of the study, i.e., can emulators be designed for the six target machines using AMD bit slice LSI components, has clearly been answered in the affirmative.

The specific conclusions of the emulator feasibility study are listed below:

A. A host machine can be designed using AMD bit slice components along with the Am29116 CPU which can support emulators of the six target machines.

B. Emulators for the six target machines can be designed employing an 80-bit (approximately) micro-control word. The emulators have been estimated to require from 300 to 400 micro-instructions.

C. The micro-instructions for the six emulators could be stored in a ROM of from 2000 to 4000 words (80-bit).

D. The generation of the micro-instructions to interpret the MIS language on the host machine could lead to a 50% reduction in the cost of emulating the six types of target machines. Further study is required to refine this estimate.

E. The cost of generating the emulators would be $250,000.00 plus the cost of developing a micro-assembler and simulation support tools. These are estimated to cost $100,000.00.

F. The cost of generating and maintaining the emulators could be reduced significantly if they were expressed in terms of a high level language and then compiled into the micro-instructions required to interpret the target machine language instructions.

The findings of this study are preliminary in nature and much more detailed information about the machine language instruction sets and internal register layouts (especially for target machines D, K, and F) will be required to insure that adequate facilities are available in the proposed emulator hardware to accomodate this equipment.

While it is believed that the 100-nanosecond internal processing speed of the Am29116 is ample to provide target machine language instruction execution times equal to or shorter than the equivalent times associated with the six microprocessors, this must be firmly established by timing studies. It also may be possible to reduce the width of the control word from the present 80-bit size. This would require careful analysis of what control lines are in the same compatability class, i.e., are never energized in the same control word, and introduce decoding switches to reduce the number of control word bits required to drive these lines.

REFERENCES

1. Brachman, R. J. , "Commonality Study of Computer Software Resources in AAH," Report 79-04, Section 104, R. J. Brachman Associates, Inc.

2. Brachman, R. J., "Commonality Study of Computer Hardware Resources in the AAH," Report 79-105, R. J. Brachman Associates, Inc.

3. Kelly, S. A. , et. al., "Master Instruction Set," R. J. Brachman Associates, Inc., December '79.

4. Harmon, B., "Bipolar Processor Makes a Powerful 16-Bit Microprogrammable Controller," Electronic Design, 21, October 11, 1979.

5. Harmon, B., Miller, W. K., "A High-Performance 16-Bit Bipolar Microprocessor -- The AM29116," Advanced Micro Devices, Sunnyvale, CA 94086.

6. Davies, P. M., "Readings in Microprogramming," IBM Systems Journal, Vol. 11 (1972), pp. 16-40.

7. Daly, E. B., "Organizing For Successful Software Development," Datamation, Vol. 25, No. 14, December 1979, pp. 107-120.

8. Putnam, L. H., Fitzsimmons, A., "Estimating Software Costs," Part I, Part II and Part III, Datamation, September, October, and December 1979.

9. Merwin, R. E., "Development of Experimental Compilers to Generate Emulators for the BMD DDP Test Bed from High Level Languages," Final Report, 1 April 1979, U.S. Army BMDATC Contract No. DASG60-78-C-0115.

10. "PASCAL to Microcode Compiler Development," Final Report, September 1979, TRW, Inc., Subcontract No. HO 7868AF9S.

11. Hewlett Packard, "Microporgramming 21MS Computers, Operating and Reference Manual," Manual No. 02108-90008, August 1974.

12. Interdata, "Model 8/32 Micro-Instruction Reference Manual," Publication No. 29-438, 1975.

APPENDIX B:

REPORT OF A TASK STUDY ON AUTOMATIC TRANSLATION OF PROGRAMS FROM ONE COMPUTER TO ANOTHER FOR THE U.S. ARMY ADVANCED ATTACK HELICOPTER FIRE CONTROL SYSTEMS

INVESTIGATORS: DR. NOAH S. PRYWES

CIHAN TINAZTEPE

KANG-SEN LU

ASSOCIATES, INC.

This page was left blank intentionally.

APPENDIX B

REPORT OF A TASK STUDY ON AUTOMATIC TRANSLATION OF PROGRAMS FROM ONE COMPUTER TO ANOTHER IN U.S. ARMY ADVANCED ATTACK HELICOPTER (AAH) FIRE CONTROL SYSTEMS

1. SUMMARY OF PROBLEM AND ALTERNATIVE APPROACHES

This study of automatic translation of computer programs from one computer to another was conducted in the context of the Advanced Attack Helicopter (AAH) Fire Control Subsystem. Presently the Fire Control System is designed using 14 embedded microprocessors of 9 different types, each programmed to perform an individual task. These programs have been developed in the assembly languages for the respective microprocessors. They amount cumulatively to approximately 200,000 lines of assembly language code. The large number of computer types and computing languages would make future maintenance, modifications and improvements very difficult and expensive. The U.S. Army is considering replacement of these embedded microprocessors by a single microprocessor. We refer to it in the following as the standard-microprocessor. Its instruction set is referred to as the master-instruction-set. Reprogramming of the respective programs, manually, using the master instruction set, would also require extensive testing for verification of the operation of the entire system, as very likely there will be differences due to the reprogramming effort.

Two approaches have been envisaged to solving this problem. The first approach, which is a subject of a separate study task, is that of emulation; namely incorporating in the standard microprocessor micro code for the instructions of the respective microprocessors, and then directly executing the original programs. We will not refer to this approach as it is the subject of a separate study task.

The other approach consists of creating a software system which will automatically translate the source assembly language programs of the respective microprocessors into the standard-microprocessor master-instruction-set language.

The present study is concerned with using an automatic translation system to generate programs, in the language of the master-instruction-set, based on the assembly language programs of the original respective microprocessors.

The requirements of such translation are quite severe, as follows:

1) An <u>all</u> automatic translation process should apply to the overwhelming majority of the source programs (say about 90%). Otherwise if extensive manual intervention is required then the possibility of introducing errors arises and a thorough verification of the system will still be needed.

2) The automatically generated object programs must be highly efficient in use of memory space as well as in execution time so that the replacement does not contradict real-time rules. It is assumed however that the standard-microprocessor is considerably more powerful than the microprocessors that it replaces.

3) In the process of translation it would be necessary to generate also documentation for the programs, to facilitate future maintenance activity. It is assumed that the source programs are presently not adequately documented.

Figure 1 is a schematic diagram of the information flow in the translation of a source assembly language program, for a respective microprocessor, into an object standard-microprocessor machine language program, with program documentation being generated as a by-product. Figure 1 portrays two alternative approaches.

The input to the translation process is an individual source assembly language program for a respective microprocessor shown on the left of Figure 1. The first step is common to both alternatives. It consists of a translator that accepts the source assembly language, program for any one of the nine different microprocessor types, and produces a <u>uniform-tabular-representation</u> for the respective program. For each source code statement there would be an entry in the table indentifying the operation, the operands and their addressing modes, the locations of instructions and data in the original microprocessor and the registers that are effected by the operation (e.g. overflow etc.). As shown in Figure 1, the translator would reference a specification of each of the respective source assembly languages in the translation process.

Alternative I consists of translation of the uniform-tabular-representation of the program into a program using the

the master-instruction-set language. Alternative II consists of a "reverse complex" to translate the uniform-tabular-representaticn into a High Level Language, such as Fortran. Both alternatives require employing complex software methods. Alternative II would generate superior documentation of the program and will be completely machine independent. However Alternative II is more difficult to achieve due to the machine independence related restrictions existing in a High Level Language. In both alternatives, the programs that are generated by the translators must be further processed by additional language processors, an assembler in Alternative I and a compiler in Alternative II. to produce the standard micro-computer language program.

The conclusions and recommendation of the study are briefly stated in Section 2. The remainder of the report discusses the process of Alternative I in Figure 1. Section 3 describes the translator from the source assembly language to a uniform-tabular-representation. Section 4 concludes with the discussion of the translator from the uniform-tabular-representation to the master-instruction-set language. To illustrate the operation of the translator, we have designed a translator for a subset of the instruction set of the M6800 microprocessor into the Z80 microprocessor. The assembler for the standard-microprocessor shown in Figure 1 is not discussed here as it is assumed to be available from the manufacturer of the standard microprocessor. As will be indicated the design of an Alternative II system poses several very difficult problems, in addition to the problem areas inherent in Alternative I which are common to both alternatives. This is one of the reasons why we recommend postponing Alternative II and why it is not discussed in detail in this report.

2. SUMMARY OF CONCLUSIONS AND RECOMMENDATIONS

This section discusses the advantages, disadvantages and risks associated with the two alternatives and the individual processes portrayed in Figure 1. While advantages and disadvantages can be stated in terms of the functions of the respective processes, risks are associated with major problems that much be solved and with limitations that may have to be imposed on the ability of the system to translate certain classes of assembly programs. As already indicated previously, at best we expect that the system would be able to translate the great majority of assembly programs. There will however, always be some programs that it would be impossible to translate fully automatically. The risks are also associated with estimates for the technical manpower that would be required for solving certain problems.

We foresee two advantages of Alternative II over

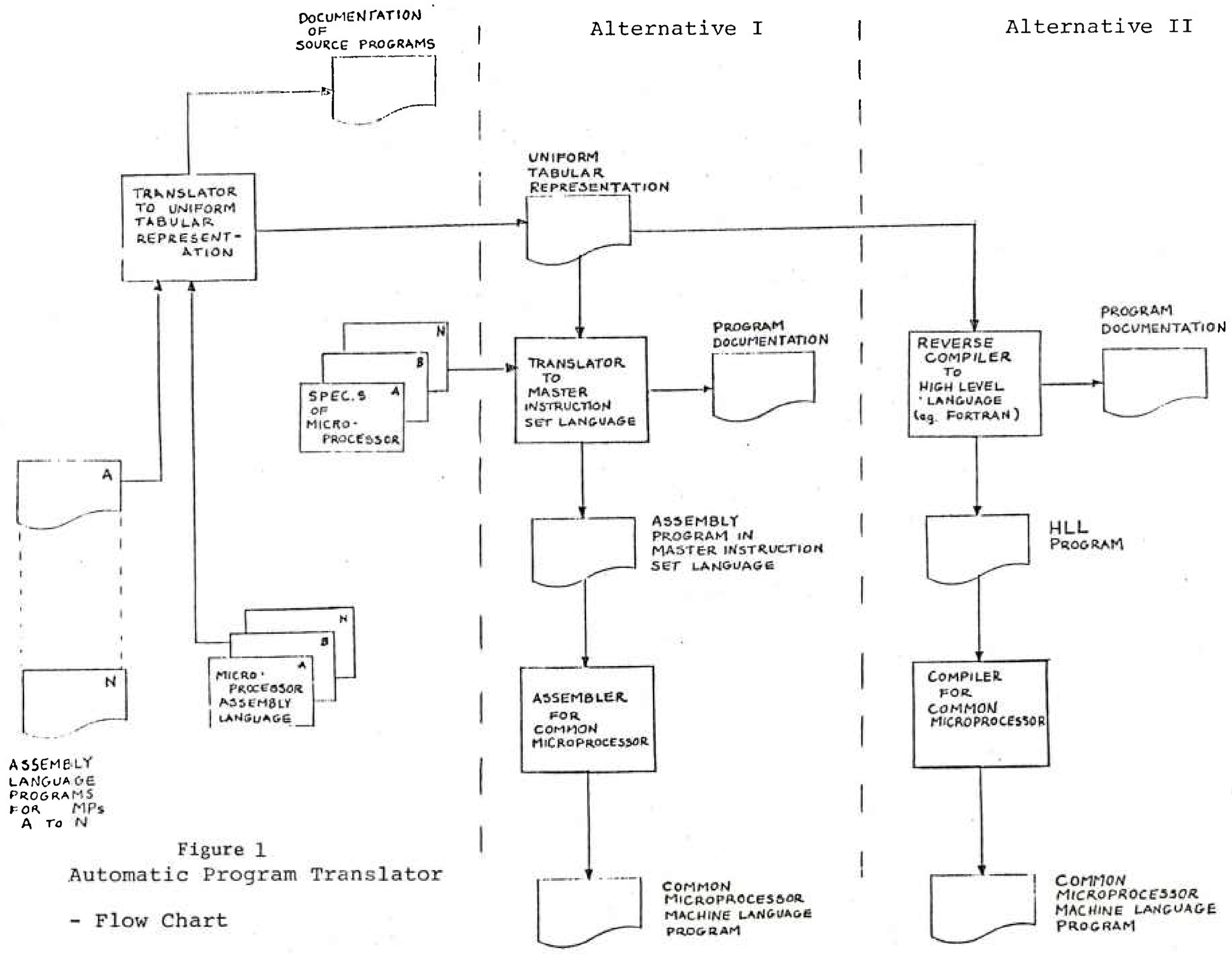

DOCUMENTATION OF SOURCE PROGRAMS
Alternative I
Alternative II
TRANSLATOR TO UNIFORM TABULAR REPRESENT-ATION
UNIFORM TABULAR REPRESENTATION
SPEC.S OF MICRO-PROCESSOR
A
B
N
TRANSLATOR TO MASTER INSTRUCTION SET LANGUAGE
PROGRAM DOCUMENTATION
REVERSE COMPILER TO HIGH LEVEL LANGUAGE (eg. FORTRAN)
PROGRAM DOCUMENTATION
A
ASSEMBLY PROGRAM IN MASTER INSTRUCTION SET LANGUAGE
HLL PROGRAM
N
MICRO-PROCESSOR ASSEMBLY LANGUAGE
ASSEMBLER FOR COMMON MICROPROCESSOR
COMPILER FOR COMMON MICROPROCESSOR
ASSEMBLY LANGUAGE PROGRAMS FOR MPs A TO N
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM
COMMON MICROPROCESSOR MACHINE LANGUAGE PROGRAM

Figure 1
Automatic Program Translator
- Flow Chart

ASSOCIATES, INC.

Alternative I as follows:

1) The High Level Language would represent a better documentation of the program then would be possible to achieve based on the master-instruction-set assembly-language. This is conditional however on success in generating a High Level Language program that is considerably shorter than the equivalent source assembly program, which is indeed a very difficult task. The advantage in documentation is derived from the fact that, a program is described in a High Level Language on a much higher level, omitting much detail that is concerned with machine level implementation of the program. Therefore it would be easier to understand and also to modify it.

2) The representation of the assembly language program by a High Level Language program eliminates all the machine dependent aspects of the implementation. Therefore it would be readily possible to transport the respective program to run on other machines in the future. While this is not an immediate requirement it may prove valueable in the long run.

The machine independence achieved in the High Level Language is also the source of major difficulties in accomplishing the translation. It is likely that only a greatly restricted class of programs can be "reversed compiled" as compared with Alternative I. The designers of High Level Languages have intentionally eliminated all features in the languages that would allow specification of physical implementation of the respective computations. For instance, the following operations, which are used by assembly language programmers, <u>cannot be stated in a High Level Language</u>.

1) High Level Language programming distinguishes between the program and the data areas in the memory and does not allow the specifying of execution of program instructions in the data area.

2) It is not possible to specify in a High Level Language computing physical addresses, of the instructions or the data. Indexing is allowed in the data area only. Thus in many instances the use of index registers, indirect addressing and other computing of addresses, widely used in an assembly language, cannot be expressed in a High Level Language.

3) The methodology used in the arithmetic unit of a specific computer and the conditions and flags used in arithmetic operations cannot be referenced in a High Level Language (except through interpretation of these operations).

4) A High Level Language imposes limitations on

operations on variables with different data types, while an assembly language programmer frequently can get around such restrictions.

5) Relative position of variables cannot be stated in a High Level Language. Therefore memory in the standard-computer cannot be allocated similar to the source microprocessor.

The problems posed by these restrictions are quite severe. It would require that the translator extract the high level concept of the computation from the source assembly program. Additional information, may be required, which would have to be prepared manually and submitted to the reverse compiler.

In addition to the above problem areas the design of an Alternative II system includes all the envisaged problems in Alternative I. Alternative I therefore is far less risky than Alternative II.

We recommend implementing Alternative I first. Based on the experience gained in the development of an Alternative I system it would be possible to assess whether an Alternative II system should be further explored. We will focus here, therefore, on Alternative I only.

In order to further reduce the risks in Alternative I we recommend that the design of the system utilize to the fullest possible extent the similarities in arithmetic operations and in memory allocation between each of the nine microprocessors used in the AAH Fire Control System and the standard-microprocessor that would replace them. The arithmetic operations and number systems of each source microprocessor will be modelled in the standard-microprocessor. Further, similar structures of the memory program and data areas, of the source and object programs, would be retained as closely as possible, even if this would reduce the efficiency of the object program. Otherwise the task of translation would be far more difficult and may involve much larger inefficiencies. This would, however, be facilitated by the greater power of the standard-microprocessor and its master-instruction-set.

Further, we will exclude from the class of assembly programs, that would be translatable, those programs which incorporate operations on locations, either in the program or data areas, where these locations serve as operands of other jump or execute instructions. However we will allow the above in the limited following case. The execute or jump operation instructions located in the data area will be accepted provided the assembly language representation of these locations is supplied by the user, in addition to the source assembly program.

Referring again to Figure 1, Alternative I is divided into two processes:

1) A translator from the source assembly language of a particular microprocessor into a uniform-tabular-representation of the program.

2) A translator from the uniform tabular presentation of the source program into an assembly language program utilizing the master-instruction-sets.

The first of these processes represents well known methodology. It can be based on similar systems developed to date with which there has been considerable experience.(1,2) Thus the design and implementation of this process involves very little risk, if any at all. Systems of this type exist presently and can be readily adapted. One system of this type,(1) developed and used for several years at the University of Pennsylvania, is described in Section 3 and an example of design of a translator form M6800 to the Z80 microprocessors is given in an appendix. The work in implementing the process would consist primarily of specifying the syntax and some semantics of the nine microprocessors. We estimate that this effort would require approximately 2 man years of attention by senior computer software specialists over a period of six to nine months, plus computer time.

The second process described above represents greater risks. We have surveyed the published technical literature in this area(1 through 7) and have found relatively little directly relevant previous experience. Based on the problems that we have studied, our conclusion is that the corresponding process is practical and can be implemented. To locate and investigate the problems that may arise we designed in detail a system that translates a subset of the instructions at the M6800 microprocessor into Z80 instructions, which is reported in the appendix. The documentation produced by this process would be similar to that produced by several commercial assembly language automated flowcharting systems.(9)

Finally, in real-time sensitive programs, the execution time of the program by the standard-microprocessor will be modelled to verify that execution time will not exceed the time required by the source microprocessor. The standard-microprocessor will be generally faster than the source microprocessors. It is assumed that the system may be sensitive to exceeding maximum execution times but not sensitive to minimum execution times.(8)

The development period is estimated at 9-12 months. It will require 3 man years of effort plus the needed computer time. A major portion of this effort will be devoted to defining the correspondence of the hardware between each one of the source microprocessors and the standard microprocessor. These definitions will be referenced in the translation process (see Figure 1).

Assuming that these two processes are implemented one following the other, the total required development time would be between 1-1/4 to 1-3/4 years at the cost of approximately 5 man years of effort plus the needed computer time.

3. TRANSLATION OF ASSEMBLY LANGUAGE PROGRAMS OF 9 MICROPROCESSORS INTO A UNIFORM-TABULAR-REPRESENTATION

The first translation process analyzes the syntax and local semantics of individual statements in an assembly language program of any one of the nine source microprocessors and produces a uniform-tabular representation of the program. It is based on advanced state-of-the-art syntax analysis techniques which have proved to be invaluable. Specifically, a parser program for these assembly languages will be generated automatically. In addition to checking the statements for syntactic and some semantic errors, the generated program will also store the statements in a tabular form for later processing.

This capability exists in a number of state-of-the-art systems.(1,2) Following is a description of such a system, the Syntax Analysis Program Generator (SAPG) developed at the University of Pennsylvania.(4) The Syntax Analysis Program (SAP) for the source assembly languages will be generated automatically by the SAPG. As shown in Figure 2, the SAPG produces the Syntax Analysis Program (SAP) for analyzing assembly language statements, based on a specification of each assembly language expressed in the EBNF/WSC (Extended Backus Normal Form with Subroutine Calls) meta language.

The EBNF/WSC includes the traditional concepts of BNF. BNF uses sequences of characters enclosed in angle-brackets < > called non-terminals to give names to grammatical units, for which substitutions may be made. BNF consists of a series of production rules of the form "A::=B". "A" is a single non-terminal symbol and "B" is one or more alternative sequences of terminal or non-terminal symbols that can be substituted for A. The alternatives are separated by the meta-symbol "|". To facilitate language description, BNF was extended to EBNF with two well-known meta-symbols: [ ] representing optionality and [ ]* representing zero or more repetitions.

The specification of the source assembly language that

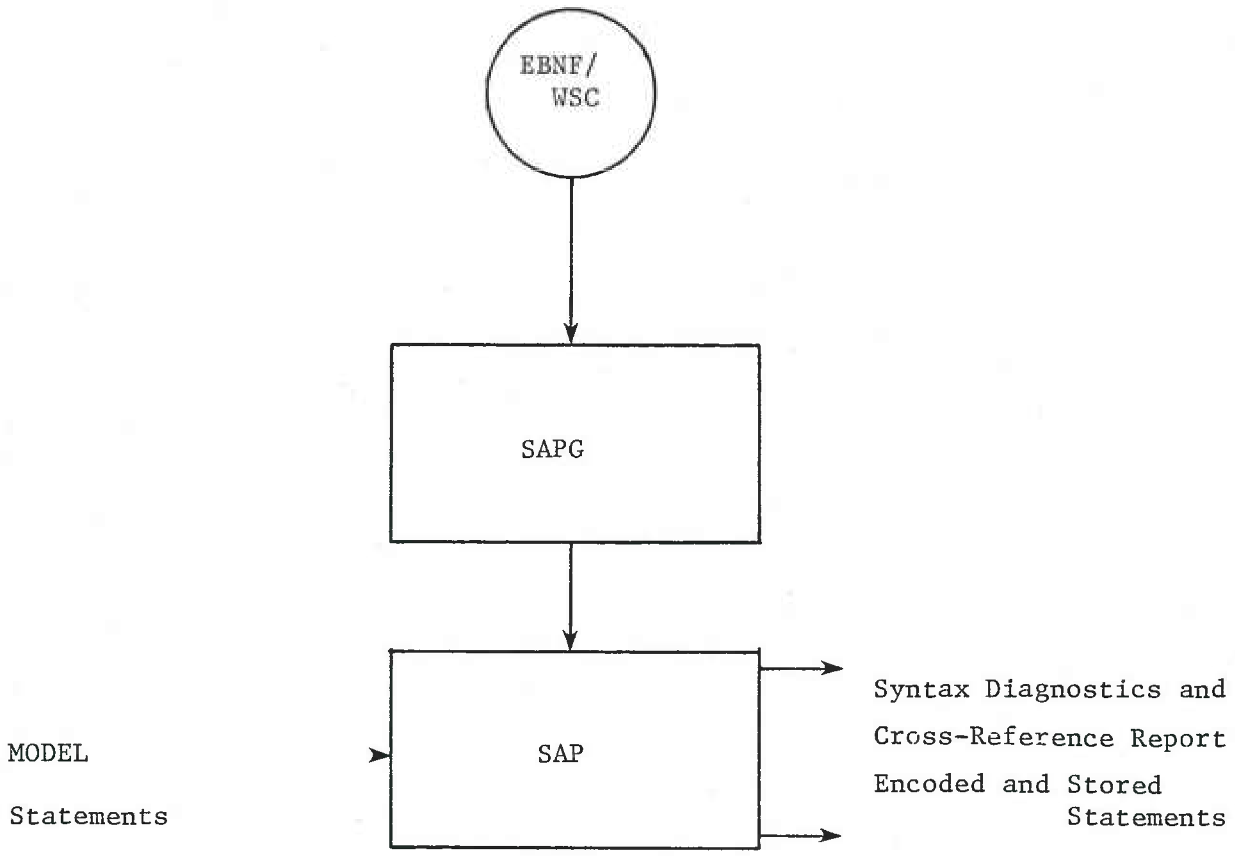


Figure 2

Block Diagram of SAPG and SAP

Repetition zero or more times is effected by generating a GO TO to the statement testing for recognition. Subroutine names embedded in the EBNF/WSC get a CALL generated for them in place. Calls to other subroutines not explicit in the EBNF/WSC are also generated. These include calls on "housekeeping" subroutines of the SAPG and calls to LEX, a subroutine to scan and return the next token in the object language. The code generated by the SAPG would become one procedure in the SAP. Note that the keywords and delimeters that the language definer uses in the production rules are preserved in the generated SAP.

A refined system flowchart of the SAPG and SAP showing the types of supporting routines appears in Figure 3. The manually-written syntactical supporting routines are of one of several types:

(1) a lexical analyzer which returns tokens of syntactic units to the SAP for analysis;

(2) statement semantics checking routines;

(3) error message handling routines;

(4) encoding routines to compact information for further efficient processing; and

(5) statement storage routines.

The purpose of the lexical analyzer is to scan for syntactic units or "tokens," using such delimeters as blanks and certain punctuation marks, and to return tokens to the Syntax Analysis Program (SAP) for syntactic checking. The automatically-generated SAP calls upon the lexical analyzer (LEX) whenever it needs the next token. The lexical analyzer is based on the finite state machine concept. Each state of the machine corresponds to a condition in the lexical processing of a character string. At each state, a character is read, an action is taken based on the character read (such as concatenating the current character to previous ones or returning the entire token to the SAP), and the machine changes to a new state. The entire character set is divided into categories such as illegal characters, delimeters, "normal" characters, etc. A state transition matrix is used. The rows of the matrix represent the character classes of the previous character, while the columns represent those classes of the current character. The entries in the matrix indicate the action to be taken and the next state. The actions involve such steps as concatenating of a character, ignoring a character, detecting an illegal character, returning a complete token to the SAP, etc., and setting a "next state."

is input to the SAPG consists not only of the syntax specification but also of subroutine names embedded within the EBNF; therefore the name "EBNF with Subroutine Calls" (EBNF/WSC). The SAPG provides a capability to branch to these subroutines upon successful recognition of a syntactic unit. Thus, they complete the SAP to enable it to check statement semantics, to encode, to produce error messages, and to store statements for later processing. The invocations of these subroutines are generated automatically by the SAPG, while the supporting subroutines themselves are written manually. The definition of a subset of M6800 microprocessors assembly language in EBNF/WSC appears in the appendix. The subroutines to be invoked are indicated between slashes (/.../). Note that subroutine calls are made after the successful recognition of syntactic units up to that point.

The SAP generated by the SAPG according to the EBNF/WSC is supplemented and linked with the routines. The SAP accepts statements in the assembly language and checks them for syntactical correctness, and local semantics. It produces a listing of the statements, syntax diagnostics, an encoded stored version of the statements, and a cross-reference report.

The SAPG is a small compiler in itself in that it processes a specification in the language EBNF/WSC and produces a program (SAP). It performs in three passes.

In pass 1. each production is scanned, and its components are encoded into a set of tables. Non-terminal symbols appearing on the left-hand-side of a production (new production names) are put into a symbol table, while non-terminals appearing on the right-hand-side of a production are put into a work table. Terminal symbols in a production are put into a terminal symbol table. Subroutine calls are put into yet another table.

In pass 2, the symbolic references in the work table (i.e. non-terminals on the right-hand side of the original production) are resolved. Pass 2 checks that each right-hand-side non-terminal symbol in the work table is defined, and links it to the corresponding entry in the symbol table. Undefined non-terminals as well as circularly-defined non-terminals can be detected in these table searches.

Pass 3 of the SAPG is the code-generation phase that produces the SAP in PL/1. It is only entered if no errors were encountered in the previous phases. For each EBNF/WSC production, a PL/1 procedure is generated. Each one returns a bit: 1 if the recognition was successful; 0 if it was unsuccessful. The exclusive nature of EBNF production rules and alternatives is effected by generating nested PL/1 IF-THEN-ELSE statements.

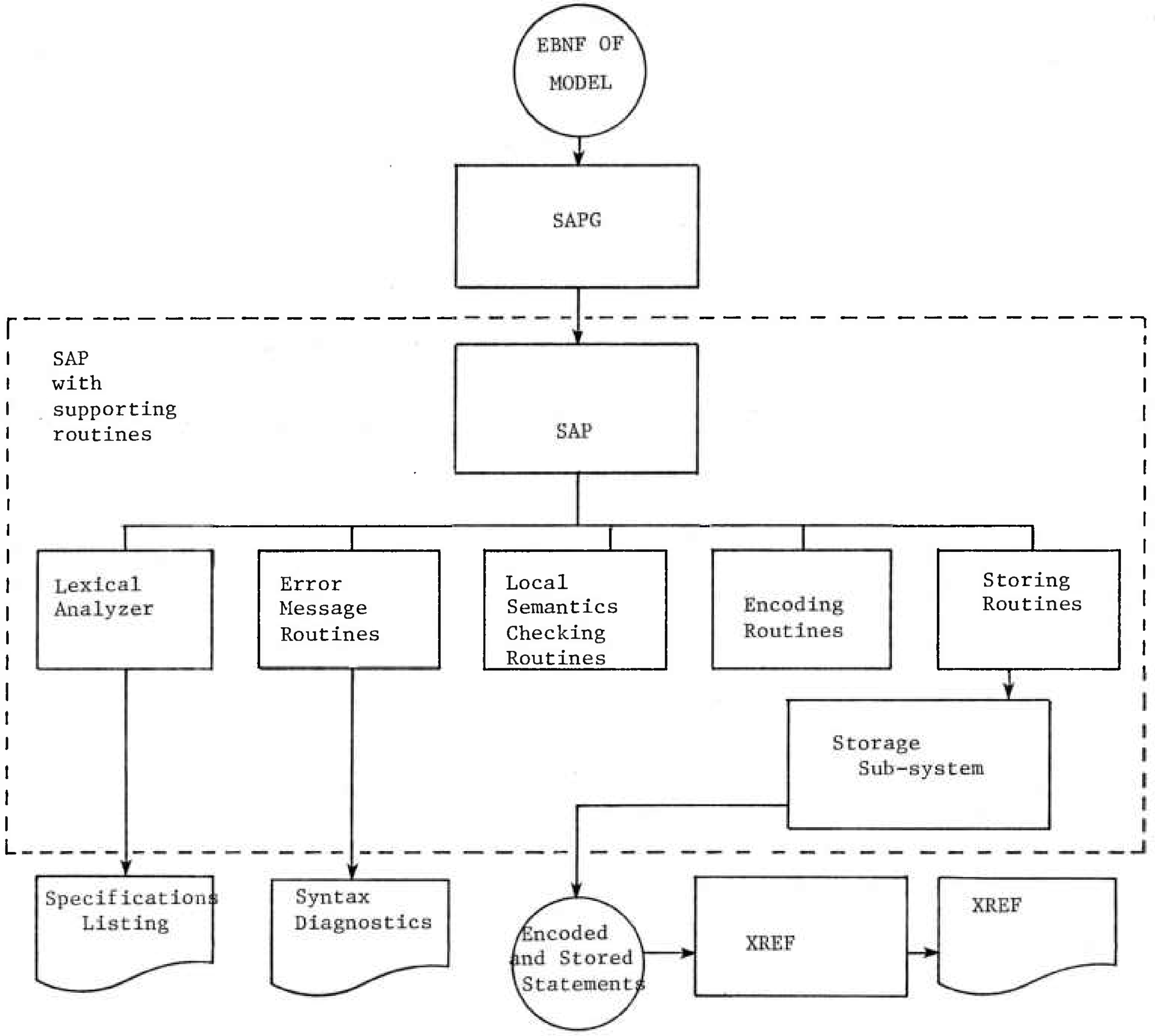


Figure 3

More Detailed View Of SAPG and SAP With Supporting Subroutines

Some of the semantics of the specification statements can be checked by the routines. An example of a local semantics checking routine is one which checks the memory locations. These manually-written routines are invoked automatically by the SAP by virtue of their specification in the EBNF/WSC.

Error subroutines stack error diagnostics to print out upon recognition of a syntactically-incorrect statement. Upon reaching an incorrect syntactic unit, the automatically generated SAP does not print its own messages, but expects the corresponding diagnostics to be on an "error stack." For this purpose, subroutines have to be written to give a user effective information when statements are incorrect. Specifically, an error message has to be stacked for each expected terminal symbol in the assembly language in case the token is missing or incorrect. Upon reaching incorrect syntactic units, the automatically generated SAP does not print its own messages, but expects the corresponding diagnostics to be on an "error stack." For this purpose, subroutines have to be written to give a user effective information when statements are incorrect. Specifically, an error message has to be stacked for each expected terminal symbol in the assembly language in case the token is missing or incorrect. If the expected token is found, the SAP simply pops the corresponding error message and continues; if the expected token is missing or incorrect, the SAP pops the corresponding error message, prints the statement number and message, scans for the end of the statement delimeter, and continues.

One product of the process is the Error Diagnostics Report containing the messages. Each message gives the diagnostics provided by the error routine and provides the exact location of the error so that it can be corrected and resubmitted by the user easily. If no syntax errors are found during the syntax analysis phase, a message will be sent that "NO ERRORS OR WARNINGS DETECTED." But if error diagnostics are produced, a flag is set to disable continuation of analysis beyond the syntax checking.

Encoding routines encode statements into the attributes in the uniform tabular representation. All of the names or addresses when provided in the source assembly program are kept intact in internal form for use by the object program. Many of the descriptions and attributes are however encoded for more compact and efficient processing later. One encoding routine is written for each encoded attribute. Each routine is invoked automatically after recognition of the syntactic unit by the SAP. The invocation is automatically generated as part of the SAP (by the SAPG) by virtue of its specification in the EBNF/WSC. The attributes of the tables consist of an operation, it's arithmetic function, registers effected, operand addresses with modes of addressing, location of instructions, etc.

Storage routines collect the strings of names and other encoded information for each assembly language statement, and pass them to the STORE system, which is a sub-system in itself to store the statements in a uniform-tabular-representation for later processing. Such storage-invoking routines are called at the end of scanning each statement. The storage subsystem which is called by these routines, stores the statements in the output table.

At the end of the syntax pass, we have the entire set of statements stored in a manner convenient for further analysis. The storing subroutines which invoke the use of the STORE system act as an interface between the automatically generated SAP and the second process described in Section 4. The storage system is an extension to the capabilities of the SAPG since it is general purpose in nature and is independent of the nature of the language specified, and could be used for processing other languages.

Finally, there are just a few "housekeeping" type subroutines which need not be written by the language definer because they are provided by the SAPG, but which need to be included in the EBNF/WSC.

4. GENERATION OF A PROGRAM IN THE MASTER-INSTRUCTION-SET ASSEMBLY LANGUAGE

This section discusses the second translation process shown in Figure 1, which transforms a uniform-tabular-presentation of the source assembly program into an object program in the master-instruction-set assembly language.

Our approach to this translation process can be visualized as modelling the source microprocessor, with its source assembly language program and data, in the standard-microprocessor and its master-instruction-set assembly language and data. The modelling can be further envisaged in three parts: hardware, program and data.

The modelling of the hardware is completely independent of the individual source assembly program that is being translated. This modelling activity defines corresponding arithmetic operations, registers, numbering systems, memory and input/output in the two microprocessors. This correspondence will have to be defined manually for each source and standard microprocessor pair. It will be stated in a tabular form and will be referenced by the translating process in the course of translating the program.

The correspondence between the source assembly language and data and the standard-microprocessor assembly language and data would be based on the individual program that is being translated.

The degree of correspondence in the modelling would also depend on certain characteristics in the source assembly language. We can basically distinguish three cases. The most severe case is where the source assembly language contains absolute addresses in the program area as operands of jump or execute instructions. In this case we will have to model the program area in the standard microprocessor to correspond to the program area in the memory in the source microprocessor. Wherever the space required for corresponding instructions in the standard microprocessor exceeds the space required for the corresponding instructions in the source microprocessor, it will be necessary to utilize an overflow program memory area in the standard-microprocessor memory, and insert there instructions which require more space. Also correspondence would have to be established between the data area in the source microprocessor and the data area in the standard microprocess.

A less severe case is where symbolic labels are used throughout the source assembly language programs as operands of jump instructions. In this case it is not necessary to retain a one to one correspondence between the two program areas.

The simplest case is where also the operands are referred to exclusively in symbolic form and no absolute addresses are used. In this case there is also much flexibility in allocating data areas.

This process can be visualized as consisting of five sequential phases: model definition, preliminary code translation, optimization, comparison of execution times and documentation. These phases are briefly described below.

The first phase consists of scanning and analysis of the entries in the uniform-tabular-presentation of the source program, to determine which of the above three cases applies; namely examine the addressing scheme-whether it is symbolic or requires also the computing of address values, and whether the operands are all in the data area or also in the program area. Based on this, the program and data memory areas of the source microprocessors may have to be mapped into the memory of the standard microprocessor.

The second phase consists of preliminary code generation in the master-instruction-set assembly-language. The translation is performed on each source assembly language statement (now in the uniform tabular representation). If possible, the object language instruction(s) is (are) placed in the area in memory corresponding to the respective source language instruction, if the object language instructions require more space (especially where there are micros) then a jump instruction is inserted in the appropriate location and the corresponding object code is placed in a overflow area as a subroutine with a return to the next instruction. This process scans the entire source program presentation a second time. It results in a preliminary program in the master-instruction-set language. Note that generally the instructions for the source microprocessor (excluding the micros) constitute a subset of the master-instruction-sets. Also the instructions in the standard-microprocessor are on the whole more compact and perform faster then in the respective equivalent instructions of the source microprocessor. Therefore in the great majority of cases it should be possible to translate each instruction in the source program into a single instruction in the object program.

The third phase is concerned with optimization of the program obtained in the previous phase. The basic notion here is that the standard microprocessor has in most cases more memory and working registers then there are in the source microprocessor. The additional memory and registers can be traded for reducing the computation time. The basic notion here is to try wherever possible to utilize registers in place of memory addresses. This phase requires a third scan of the program in order to create a graph representation that is used to identify the scope of each iteration, each subroutine and each program branch. These subparts of the programs, constitute subprograms which will be individually optimized. The global variables of the program which are used to communicate between the above mentioned subprograms should be retained in main memory. Variables which are local to the subprograms can be moved to the register and thereby reduce the needed memory area. Also sequences of instructions in the source program may be replaced by a single of few more powerful instructions of the master-instruction-set. This may reduce both the number of instructions (and execution time) and the program memory area.

Note that the correspondence of individual subprograms in the source program and the object program must be retained.

Based on this graph, it will be possible in the next phase to scan the subprograms in the source and object programs and compute execution times for these subprograms. The comparison of overall execution times would be possible in some instances giving a clear indication whether maximum execution time requirements for a real time system will be met by the replacement standard microprocessor. In other cases where execution times are data dependent it will be possible to present to the user only comparisons of performance times for the subprograms. Further analysis would then be required by the user to determine whether the replacement-microprocessor will meet real time requirements.

The final phase is concerned with generating a documentation of the program. As already indicated the approach that we propose is essentially to use the techniques that are incorporated in a number of commercial assembly language automatic flowcharting systems. The experience to date is primarily in flowcharting assembly language programs for the IBM system 370.[9] The documentation will also include cross reference listings and data field analysis which will aid in program maintenance and modification. To obtain a more readible flowchart the individual assembly language operation may be expanded to equivalent English words. Vectors of data which are scanned in iterations may be also identified by respective iteration instances. The user would also have an option to obtain a full flowchart showing each instruction, or ultimately grouping the instructions in each of the subprograms (identified in phase 2) as a single entry in the flowchart. An edited listing of the assembly language program will also be produced with comments identifying each of the subprograms.

APPENDIX

In the following example, a SAPG generated program will be used for the assembly language translation from the M6800 to Z80 microprocessor. This is an automatic translator for the cases where the references to real addresses do not effect the translation.

The principle of translation is parsing each source assembly instruction and calling appropriate semantic routines to generate corresponding object assembly language instructions for it.

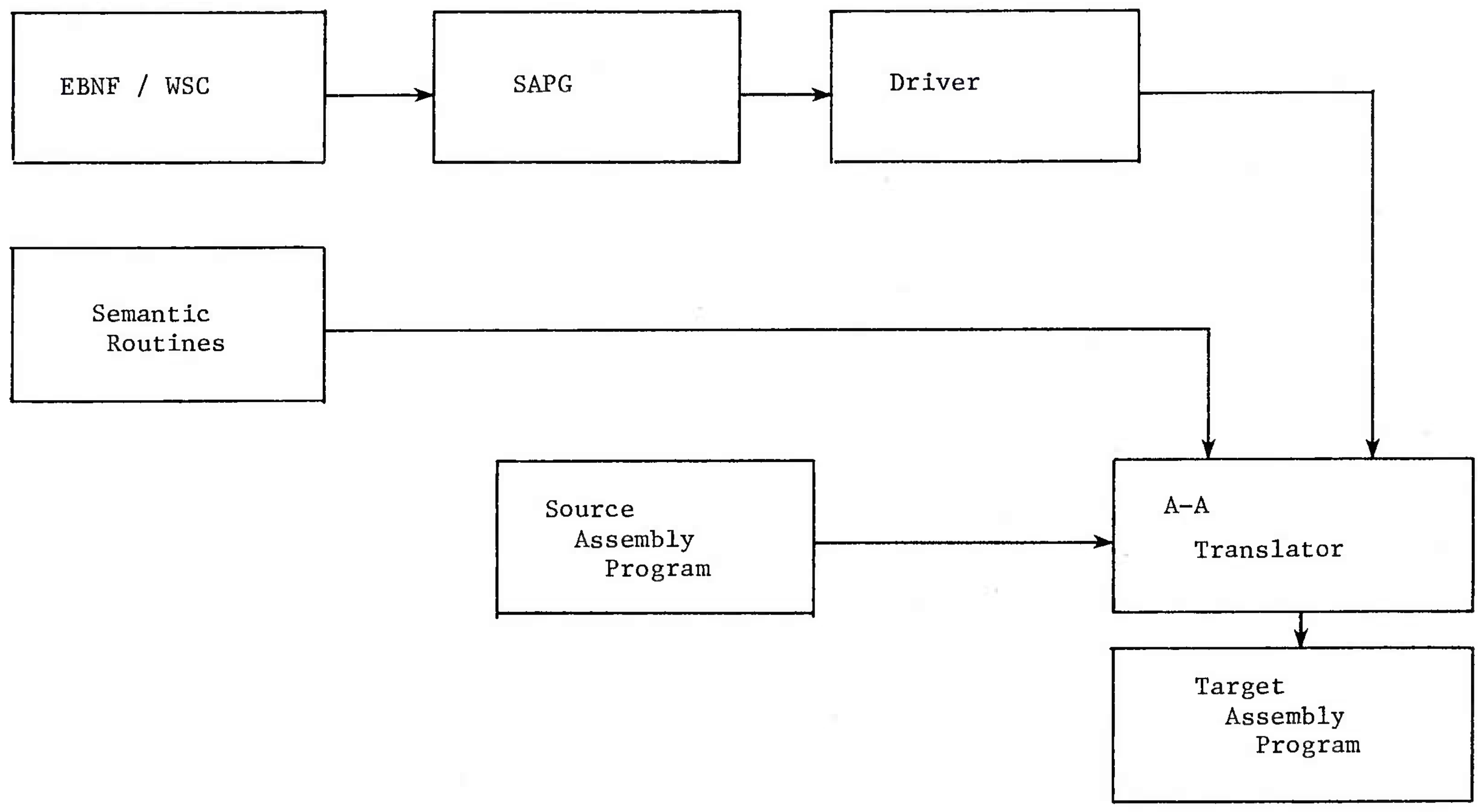
The SAPG program will accept a set of syntax rules which describes the syntax of the source assembly language (including macros). The syntax rules include semantic routine calls. The output of SAPG will be a driver program which parses the source assembly program and calls on a set of manually prepared semantics routines to generate object assembly instruction sequences.

We have to write a set of syntax rules for source assembly language using EBNF/WSC and prepare a set of semantic routines.

The source machine is M6800.

The target machine is Z80.

For the sake of simplicity we chose a subset of the M6800 instruction repertoire.



The block diagram of an M6800 assembly to Z80 assembly translator

The instructions subset for the M6800 is listed as follows:

ADDA	$A+M \rightarrow A$
ADDB	$B+M \rightarrow B$
INC	$M+1 \rightarrow M$
INCA	$A+1 \rightarrow A$
INCB	$B+1 \rightarrow B$
DEC	$M-1 \rightarrow M$
DECA	$A-1 \rightarrow A$
DECB	$B-1 \rightarrow B$
LDAA	$M \rightarrow A$
LDAB	$M \rightarrow B$
STAA	$A \rightarrow M$
STAB	$B \rightarrow M$
SUBA	$A-M \rightarrow A$
SUBB	$B-M \rightarrow B$
CMPA	$A-M$
CMPB	$B-M$
BCC	$C=0$
BCS	$C=1$
BEQ	$Z=1$
BMI	$N=1$
BVS	$V=1$
JMP	
JSR	
RTS	

The syntax rules for this instruction subset are as follows:

```
< assembly-program > ::= [ <instruction> ]*

< instruction > ::= /RESET_LABEL/ [< label_check >< NAME >
                    /SAVE_LABEL/]
                    BODY   /END_LINE/

< label_check > ::= /ANY_LABEL/

< BODY > ::= <ADD> | <INC>|<DEC>|<LDA>|<STA>|<SUB>
             |<CMP> | <BRA>|<JMP>|<JSR>|<RTS>

< ADD > ::=  ADD < TWO_OPERAND > /GADD/

< two_operand > ::=  /SAVE_OP/  < A_OR_B>,<OPERAND>

< A_OR_B > ::= A/SAVE_REGA/ | B/SAVE_REGB/

< LDA > ::= LDA < TWO_OPERAND > /GLDA/

< STA > ::= STA < TWO_OPERAND > /GSTA/

< SUB > ::= SUB < TWO_OPERAND > /GSUB/

< CMP > ::= CMP < TWO_OPERAND > /GCMP/

< INC > ::= INC < ONE_OPERAND > /GINC/

< ONE_OPERAND > ::= /SAVE_OP/ /RESET _AB/ < ONLY_OPERAND >

< ONLY_OPERAND > ::=  <A_OR_B>|<OPERAND>

< DEC > ::= DEC < ONE_OPERAND > /GDEC/

< BRA > ::=<BRA_CODE>  /SAVE_OP/ < OPERAND > /ASSBRA/

< BRA_CODE > ::= BCC|BCS|BEQ|BMI|BVS

< JMP > ::= JMP < OPERAND > /ASSJMP/

< JSR > ::= JSR < OPERAND > /ASSJSR/

< RTS > ::= RTS /ASSRTS/

< OPERAND > ::= < NAME >/SAVE_OPD_NAME/|< NUMBER >/SAVE_OPD_NUM/
```

There are nine global variables:

```
1) DCL   HAS_LABEL BIT(1);   /* 0=NO, 1=HAS */
2) DCL LABEL CHAR(6);   /* STORE LABEL */
3) DCL OP_CODE CHAR(3);   /* STORE OP_CODE */
4) DCL HAS_REG BIT(1); /* 0=NO, 1=HAS */
5) DCL REG CHAR(1); /* A OR B */
6) DCL KIND_OPD FIXED BIN; /*  1=NAME, 2=NUMBER */
7) DCL OPD_NAME CHAR(6): /* SYMBOL */
8) DCL OPD_NUM FIXED BIN; /* IMMEDIATE DATA */
9) DCL INDEX_USE CHAR(1); /* 'X' OR 'b' */
```

All the semantic routine can be defined as follows:

```
1) RESET_LABEL:  PROC;
       HAS_LABEL = 'Ø'B;
   END RESET_LABEL;
2) ANY_LABEL:  PROC RETURNS (BIT(1));
       RETURN(LINEBUF(1)  ¬='b');
   END ANY_LABEL;
3) SAVE_LABEL:  PROC;
       LABEL = LEXBUFF;
   END SAVE_LABEL;
4) SAVE_OP:  PROC;
       OP_CODE = LEXBUFF;
   END SAVE_OP;
```

```
5) SAVE_REGA:  PROC;

       HAS_REG = '1'B;

       REG = 'A';

   END_SAVE REGA;

6) SAVE_REGB:  PROC;

       HAS_REG = '1'B;

       REG = 'B';

   END SAVE_REGB;

7) RESET_AB:  PROC;

       HAS_REG='Ø' B;

   END RESET_AB;

8) SAVE_OPD_NAME:  PROC;

       KIND_OPD = 1;

       OPD_NAME=LEXBUFF;

   END SAVE_OPD_NAME;

9) SAVE_OPD_NUM:  PROC;

       KIND_OPD = 2;

       OPD_NUM = CONVERT (LEXBUFF);

   END SAVE_OPD_NUM;
```

Since Z80 has only one accumulator (REG A), all the arithmetic and logic operations have to be done in it. But in M6800, there are two general purpose registers (ACCA and ACCB). We cannot assign the only accumulator in Z80 to either ACCA or ACCB. So we will assign the REG B and REG C in Z80 to store the value of ACCA and ACCB respectively.

In Z80 there are two index registers (IX and IY), we can arbitrarily assign IX to store the value of IX in M6800.

In M6800, there are five different addressing modes, we will define the corresponding instruction sequence in Z80 for each of these addressing modes.

1) Immediate mode: In Z80, this is also an implemented addressing mode, so there is no problem to simulate.

2) Direct & Extended mode: In Z80, only LD instruction allows direct addressing mode. So for all other instruction we have to load the address into HG register pair, then use HL as pointer which points to the operand stored in memory.

3) Index mode: In Z80, it is also implemented.

4) Relative mode: In Z80, it is implemented, but not complete so we will use direct addressing mode to replace it and then treat it as Direct & Extended mode.

```
GADD:   PROC;
        IF REG = 'A' THEN GENERATE('LD A,B');
                     ELSE GENERATE('LD A,C');
        IF INDEX_USE = 'X' THEN
        DO: /* INDEX MODE */
            GENERATE('ADD A,(IX '||OPD_ADDR||')');
        END;
        ELSE DO;
            IF KIND_OPD=2 THEN GENERATE('ADD A, '||OPD_NUM);
            ELSE DO; /*NOT IMMEDIATE MODE */
                GENERATE('LD HL, '||OPD_NAME);
                GENERATE('ADD A,(HL)');
            END;
        END;
        IF REG='A' THEN GENERATE('LD B, A');
                   ELSE GENERATE('LD C, A');
        END GADD;
```

Simimlarly, we can implement GSUB, GCMP.

```
GLDA:   PROC;
        IF INDEX_USE = 'X' THEN
        DO: /* INDEX MODE */
            GENERATE('LD A,(IX '||OPD_ADDR||')');
        END;
        ELSE DO:
            IF KIND_OPD=2 THEN /* IMMEDIATE MODE */
            GENERATE('LD A, || OPD_NUM);
            ELSE /* DIRECT OR EXTENDED MODE */
            GENERATE('LD A,(' || OPD_NAME || ')');
        END;
        IF_REG = 'A' THEN GENERATE('LD B,A');
                     ELSE GENERATE('LD C,A');
        END GLDA;
```

Similarly GSTA can be implemented.

```
ASSBRA:   PROC;
          DCL COND CHAR(2);
          IF OP_CODE = 'BCC' THEN COND = 'NC';
          ELSE IF OP_CODE = 'BCS' THEN COND = 'C';
          ELSE IF OP_CODE = 'BEQ' THEN COND = 'Z';
          ELSE IF OP_CODE = 'BMI' THEN COND = 'M';
          ELSE IF OP_CODE = 'B VS' THEN COND = 'PE';
          GENERATE('JP' || COND ||', '||OPD_ADDR);
END ASSBRA;

ASSJMP:   PROC;
          GENERATE('JP '||OPD_ADDR);
END ASSJMP;

ASSJSR:   PROC;
          GENERATE('CALL 'CALL '||OPD_ADDR);
END ASSJSR;

ASSRTS:   PROC;
          GENERATE('RET');
END ASSRTS;
```

```
GINC:   PROC;
        IF INDEX_USE = 'X' THEN
           GENERATE('INC (IX+'OPD_ADDR ')');
        ELSE IF  HAS_REG THEN
           DO;
               GENERATE('LD HL, '||OPD_NAME);
               GENERATE('INC (HL)');
        END;
        ELSE IF REG = 'A' THEN
            GENERATE('INC B');
            ELSE GENERATE('INC C');
        END GINC;
```

Similarly GDEC can be implemented.

REFERENCES

1. A. French, "A Syntax Analysis Program Generator," MS Thesis, Dept. of Computer and Information Science, University of Pennsylvania, 1972.

2. W. A. McKeeman, J. J. Horning and D. B. Wortman, "Compiler Generator," Prentice Hall, 1970.

3. J. R. Wolberg & A. Peled, "Using Convert to Transform Source Code," Technion, Haifa, Israel, 1976.

4. J. R. Wolberg & A. Peled, "Convert-A Language for Program and Data File Conversion," Technion, Haifa, Israel, 1976.

5. George C. Hopkins, "Convert - An IBM to CDC Program Conversion Code," Los Almos Lab., October 1970.

6. A. J. Korenjak, "A Study in Program Conversion," Applied Logic Corp., Princeton, N.J. 1970.

7. P. Barbee, "The Filer System of Computer Program Translation," Frobe Consultants Inc., Phoenix, Arizona, 1974.

8. Christan Cesar, "Real Time Emulation of Hardware," Ph.D. Dissertation Department of Computer and Information Science, University of Pennsylvania, 1979.

9. Applied Data Research Inc., "Autoflow II," Princeton, N.J.

APPENDIX C:

REPORT OF A TASK STUDY

DEVELOPMENT OF A

MASTER INSTRUCTION SET (MIS)

FOR THE

U.S. ARMY ADVANCED ATTACK HELICOPTER FIRE CONTROL SYSTEM

INVESTIGATOR: DR. SUSAN A. KELLY

ASSOCIATES, INC.

TABLE OF CONTENTS

SECTION 1

MASTER INSTRUCTION SET

1.1 INTRODUCTION

The AAH has many microprocessor-based systems which basically utilize nine different microprocessors. Each microprocessor requires its own development and support systems. It would be better if this redundancy was eliminated by concatenating the nine different instruction sets into a superset. Unfortunately, the microprocessor which executes this super instruction set does not exist. A simpler solution would be to consolidate the various instruction sets, thereby eliminating redundancy, and resulting in a Master Instruction Set (MIS). This report describes the theoretical requirements for a microprocessor which executes the MIS. In addition a machine by machine translation from the native instruction set to MIS is provided.

1.2 ASSEMBLER REQUIREMENTS

To accomplish the goal of consolidating the various instruction sets, a cross-assembler is required which takes the original source code and translates it into coding for the MIS. This assembler would have to be a macroassembler since some instructions may be more economically translated as a sequence of instructions rather than microcoding involved commands. The need for macro capability is also dictated by the fact that the source code may also contain macros.

The operation of this theoretical macroassembler is in two stages. First, the source machine is specified and a line by line translation is produced. Second, the new translated code will be assembled into machine code.

In addition to the "ordinary" commands in the MIS there is a separate class, called the operate or OP-class. In essence, these are just miscellaneous instructions. They are unique to individual microprocessors and therefore did not warrant separate instructions in the MIS. This report provides the macro coding for each of the OP-class instructions. When the actual microcoding of this heretofore theoretical machine is performed, it is entirely feasible that these instructions might also be microcoded, but in any event, the OP-class instructions must be provided for.

Finally, if an advanced arithmetic chip (such as the 9511A) is used in conjunction with the microprocessor (29116) then certain instructions such as SIN, EXP, and SQRT, may become available to speed processing even though they are not presently included in the Master Instruction Set. In this case there will have to be an intervention by a software engineer to hand-substitute these commands for the blocks of code accomplishing the same functions.

SECTION 2

PROGRAMMING MODEL

2.1 INTERNAL ARCHITECTURE

The 29116 is a 16 bit machine with 32 internal registers. For the MIS, the lower 16 registers R_0 through R_{15} are unassigned and are available as general purpose registers. In the instance where individual microprocessors have registers with specific names, they are assigned to registers R_0 through R_{15} as required. For example, many microprocessors have a register called an accumulator. This might be assigned register R_0. All assignments are given for each individual machine in sections 4-11.

The upper 16 registers have specific assignments. The following is a list of these registers and their assignment according to the MIS.

Register	Abbreviation	Assignment
16	PC	Program Counter
17	PSW	Processor Status Word
18	IM	Interrupt Mask
19	SP	Stack Pointer
20	CRU	Communications Register
28	ER	Extension Register
30	Q	Quotient Register

Both RAM and ROM are assumed to be 16 bits wide. All addresses will be 16 bits long obviating the need for modes of addressing such as direct or zero-page.

2.2 FLAGS

The PSW is considered to be an aggregate of 16 different flag bits as illustrated in Figure 1. When the flag bit is 1, the condition that flag represents is considered true as of the last operation. It is assumed that the processor will set or clear flags as required. The definition of each bit is described below.

Bit	Abbreviation	Definition
Ø	C	8 bit carry
1	V	8 bit overflow
2	Z	8 bit zero or equal
3	N	8 bit negative bit
4	P	Even parity check
5	H	8 bit half-carry
6	>Ø	Greater than zero
7	≥Ø	Greater than or equal to zero
8	A>	Arithmetically greater than
9	L>	Logically greater than
A	1	Always set to 1

ASSOCIATES, INC.

BIT ASSIGNMENT OF PROCESSOR STATUS REGISTER

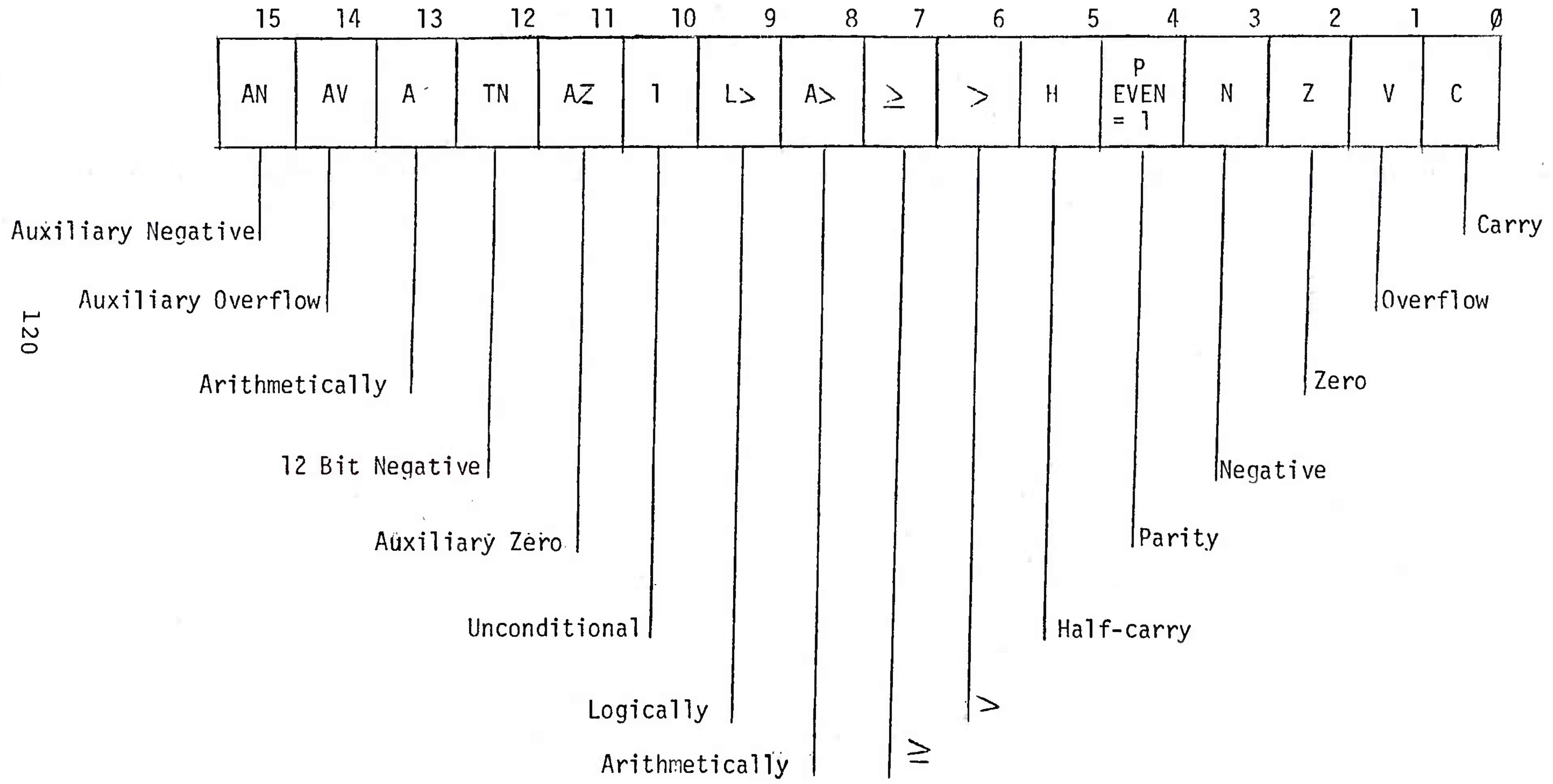


FIGURE 2.2.1

Bit	Abbreviation	Definition
B	AZ	Upper 8 bit zero
C	TN	12 bit negative
D	A<	Arithmetically less than
E	AV	16 bit overflow
F	AN	16 bit negative

Carry Bit (C) - Bit Ø is modified as a result of specific operations, such as ADCw and SBCw, or directly with commands such as SETØ and RESØ. This bit serves as either 8 or 16 bit carry depending on w.

Overflow Bit (V) - Bit 1 is set or cleared as a result of byte arithmetic operations. It will be modified during add and subtract operations when the least significant 8 bits result in a value which cannot be contained in those 8 bits. Similarly, bit E is set whenever the least significant 16 bits cannot accommodate the result of an arithmetic operation.

Zero Bit (Z) - Bit 2 is automatically set to one whenever the result of an operation equals zero. Therefore bit 2 is set to one whenever all the bits of the result are zero, and reset whenever any of the bits are not zero. Bit B performs the identical function for the upper byte.

Negative Bit (N) - Bit 3 contains the value of the sign bit (bit 7) produced by all arithmetic instructions operating upon 8 bit words. Bit C is set whenever a 12 bit result of an arithmetic instruction produces a negative result. Similarly bit F is set whenever a 16 bit result of an arithmetic instruction is negative.

Parity Bit - Bit 4 is set whenever the result of a parity check is even. Bit 4 is reset when the result of a parity check is odd.

H Bit - Bit 5 is set when a carry occurs during an ADNB or SBNB operation. The carry can then be transferred from the least significant nybble (4 bits) to the most significant nybble.

Greater Than Zero Bit - Bit 6 is set whenever a data movement or arithmetic operation produces a result $>$ zero. Bit 6 is reset to zero whenever the result is $\leq$ zero.

Greater Than Or Equal To Zero Bit - Bit 7 is set whenever a data movement or arithmetic operation produce a result greater than or equal to zero. Bit 7 is reset to zero whenever the result is less than zero.

Arithmetically Greater Than - Bit 8 is set when the result is arithmetically greater than the source with which it is being compared.

Logically Greater Than - Bit 9 is set whenever the result of a Boolean operation is greater than the source with which it is being compared.

Bit A - This bit is always one to allow an unconditional program transfer when the following instructions are executed: JMAS, RTAS, JLAS.

Arithmetically Less Than - Bit D is set when the result is arithmetically less than the source to which it is being compared.

2.3 INTERRUPTS

Interrupts provide a microprocessor with the means of detecting external asynchronous events. Generally an interrupt request is transmitted to the microprocessor via a voltage level or transition. An interrupt request may be ignored in certain instances or if it is non-maskable then it must always be serviced. A special register IM, is available for specifying whether or not an interrupt is to be serviced.

There are two philosophies in dealing with interrupts. The first specifies that if an interrupt is to be serviced, then the processor transfers control to a specific location in memory. This is a vectored interrupt. If the processor goes to a specific location and then fetches the location to which control will be transferred, it is called vector fetch. A simple way of reconciling the two types is to make this theoretical processor a vector fetch type. Then to execute instructions for a vectored type, just place the normal vectored location in the memory fetch addresses and operation will proceed as expected.

Unfortunately, the precise nature of interrupt instructions are idiosyncratic to each machine. Therefore, special instructions are required to enable the MIS processor to perform all types of interrupt service. These are the IN and RI instructions. It is assumed these will be microprogrammed for each machine appropriately.

2.4 I/O CONTROL

Communication with the external environment is generally done via input/output (I/O) ports. Some processors have specific I/O commands which transfer contents between register and I/O ports. This type of I/O operation is referred to as direct I/O. Other processors bring the full power of their instruction sets to I/O operations by treating I/O ports as memory locations. These machines have no specific I/O operations. This type of I/O processing is called memory-mapped.

It is assumed that the MIS processor will use memory-mapped I/O. When translating instructions from direct I/O machines, the MIS macroassembler will simply substitute MOV instructions to the address assigned the I/O ports.

2.5 ADDRESSING MODES

Each of the processors has a series of addressing modes associated with it. The MIS incorporates all of these addressing modes. Table 1 lists the addressing modes and the notation associated with them. Also included is a symbolic description of how the effective address is formed. Following

Table 1 is a more detailed description of the various addressing modes.

TABLE 1

SUMMARY OF ADDRESSING MODES

Symbols		Abbreviations	
( )	Contents of	I	Operand
[ ]	Effective address	M	Memory address
$\leftarrow$	"Is transferred to"	R	Register
@	Indexed	UR	Upper-half register
#	Immediate address	Off	Offset

Addressing Mode	Notation	Interpretation Of EA
Immediate	#I	I
Absolute	M	[M]
Indirect	(M)	[(M)]
Register	R	(R)
Upper Half Register	UR	$(R)_{8-15}$
Indirect via Register	(R)	[(R)]
Pre-decrement R	-R	(R)←(R)-1
Post-increment R	R+	(R), (R)←(R)+1
Indirect via Register		
a. Pre-increment	+(R)	[(R)←(R)+1]
b. Post-increment	(R)+	[(R)],(R)←(R)+1
c. Pre-decrement	-(R)	[(R)←(R)-1]
d. Post-decrement	(R)-	[(R)],(R)←(R)-1
Indexed		
a. Indexed absolute	M@R	[M+(R)]
b. Indexed indirect	(M@R)	[(M+(R))]
c. Indexed indirect Post-increment	(M@R)+	
Extended	M@R*	[M+(R*)]
Offset		
a. Relative	(R)+Off	[(R)+Off]

Addressing Mode	Notation	Interpretation Of EA
b. Indirect via R	(R+Off)	[((R)+Off)]
c. (b)+ Post-increment offset	(R+Off)+	[((R)+Off)], (R)←(R)+1
d. Index, indirect offset	M@(R+Off)	[M+ ((R)+Off)]
Assembler Directive		

Absolute (Direct) - In this mode, the address following the opcode is used as a pointer to the operand which is then fetched from memory. Generally the full 16 bit address of any memory location is specified. The 16 bit address obviates the need for paged addressing, particularly zero paged addressing. That is, there is no advantage in assuming the upper byte to be zero, since the upper byte is always included in a 16 bit address.

Indirect - In this mode the contents of the address contained in the instruction serves as a pointer to the operand.

Register - This addressing mode is similar to the absolute mode described above except that the operand is specified as the content of a Register (R).

Upper-Half Register - the operand, in this mode, is the content of the upper-half (UR) of a given Register, R.

Indirect Via Registers - In this mode the address of the operand is specified by the contents of the Register (R).

Register Modify

a. Pre-decrement register - In this mode the address of the operand is found by decrementing the contents of Register R, then using the updated contents of Register R as the operand.

b. Post-increment register - This mode is similar to the Register mode described above in that the operand is specified by contents of a given Register but then the contents of Register R are incremented by one.

Indirect Via Register

a. Pre-increment - The contents of Register R are incremented one. The address of the operand is then specified by the contents of the address pointed to by Register R.

b. Post-increment register - The effective address of the operand is specified as the content of the address pointed to by Register R. The content of Register R is then incremented by one.

c. Pre-decrement register - The content of Register R is decremented, the operand is then pointed to by the content of Register R.

d. Post-increment register - The operand is pointed to by the content of Register R. The content of Register R is then incremented by one.

Indexed

a. Indexed absolute - In this mode the effective address of the operand is found as follows: The address field accompanying the opcode is added to the contents of the specified Index Register.

b. Indexed indirect - This mode is similar to Indexed absolute except two operations are required. First the contents of Register R are added to the address field accompanying the opcode. This value points to a location which in turn points to the address containing the operand.

c. Indexed indirect post-increment - This is identical to Indexed indirect except that after the effective address is formed, Register R is incremented by one.

Extended - In this mode it is assumed that the status of the extended register specifies the field (each field contains 64K) that contains the address of the current instruction.

Offset

a. Relative - In this mode the effective address containing the operand is calculated by adding an offset value to the contents of Register R.

b. Indirect via register offset - The contents of Register R are added to the offset. This value then specifies the address which points to the address containing the operand.

c. Indirect via register offset; post-increment - This mode is identical to b, except that the contents of Register R are incremented after the effective address has been computed.

d. Indexed indirect offset - The effective address is calculated by first adding the contents of Register R to the offset. This value is then used to point to an address which is added to the address accompanying the opcode. This final value points to the address which contains the operand.

Assembler Directive

Strictly speaking, this is not an addressing mode. However, some instructions require an additional parameter for correct operation and the second operand field is utilized for this specification.

SECTION 3

MASTER INSTRUCTION SET

After consolidating all 8 instruction sets, a Master Instruction Set was produced containing 31 basic instructions. Three classes, OP, IN, RI are somewhat less defined, in that each is specific to one processor. The precise description of each of these three types is included in the machine by machine translation.

Table 2 presents a list of the symbols and abbreviations utilized by the MIS. Table 3 lists the 31 instructions of the MIS. Following this table is a detailed description of the operation of each instruction.

TABLE 2

SYMBOLS AND ABBREVIATIONS

Symbols

$\rightarrow$	Transfer to	$\wedge$	Logical and
$\leftrightarrow$	Exchange contents	$\vee$	Logical or
*	Multiply	$\veebar$	Logical exclusive or
$\div$	Divide	( )	Contents of
$\overline{SD}$	Complement	[ ]	Effective address

Abbreviations

A	Arithmetic
BCD	Binary coded decimal
b	Bit number
C	With carry
D	Destination
L	Logical
m	Mode
M_p	I/O port address
N	No carry
n	Number of times
P	Place
PSW	Processor status word
R	Register
s	Bit condition, S = set, C = clear

Abbreviations

S	Source	
SD	Operand serves as both S and D	
UR	Upper-half of 16 bit register	
w	Word size	
	B	8 bit word
	T	12 bit word
	W	16 bit word
	D	32 bit word

TABLE 3

MASTER INSTRUCTION SET

	Mnemonic	Operand(s)	Operation
1.	ADmw	S, R	$S + R[+ C] \longrightarrow R$
2.	ANDw	S, R	$S \wedge R$
3.	ANRw	S, R	$S \wedge R \longrightarrow R$
4.	CLRw	D	$\emptyset \longrightarrow D$
5.	CMPw	S_1, S_2	$S_1 - S_2$
6.	COMw	SD, k	$\overline{SD} \longrightarrow SD + k$, $k = \emptyset$ or 1
7.	CONT		No operation
8.	DAJw	SD	BCD $(SD) \longrightarrow SD$
9.	DECw	SD	$(SD) - 1 \longrightarrow SD$
10.	DIVw	S, SD	$SD \div S \longrightarrow SD$, Q (remainder)
11.	EXRw	S, R	$S \veebar R \longrightarrow R$
12.	INCw	SD	$(SD) + 1 \longrightarrow SD$
13.	INij		Interrupt command
14.	JLbs	R, P	Jump and link with R, conditional
15.	JMbs	P	Jump, conditional (bit b of PSW)
16.	JRbs	R, P	Test register and jump, conditional
17.	MOVw	S, D	$(S) \longrightarrow D$
18.	MPYw	S, SD	$S * SD \longrightarrow SD, Q$

	Mnemonic	Operand(s)	Operation
19.	OPij	S, SD	Operate class
20.	RESb	SD	$\emptyset \wedge SD_b \longrightarrow SD$
21.	ORRw	S, R	$S \vee R \longrightarrow R$
22.	RIij		Return from interrupt
23.	RLnw	SD, N or C	Rotate left n times
24.	RRnw	SD, N or C	Rotate right n times
25.	RTbs	F	Return, conditional
26.	SBmw	R, S	$R - S[- C] \longrightarrow R$
27.	SETb	SD	$1 \vee SD_b \longrightarrow SD$
28.	SLnw	SD	Shift left n times
29.	SRnw	SD, L or A	Shift right n times
30.	XCRw	SD_1, SD_2	$SD_1 \longleftrightarrow SD_2$
31.	XEQw	F	Execute contents of register specified

ADmw — Add

Operation: $S + R\ [+C] \longrightarrow R$

Description: The contents of S are added to the contents of R. The result is placed in R. The content of the carry bit can be included or omitted depending upon the mode (m); ADCw refers to addition with carry; ADNw refers to addition without carry.

ANDw — Logical and

Operation: $S \wedge R$

Description: The contents of S and R are logically ANDed. The contents of S and R remain unchanged. This instruction sets the flags of the PSW.

ANRw — Logical and with replacement

Operation: $S \wedge R \longrightarrow R$

Description: The contents of R are logically ANDed with the contents of S. The result is placed in R.

CLRw — Clear word

Operation: $\emptyset \longrightarrow D$

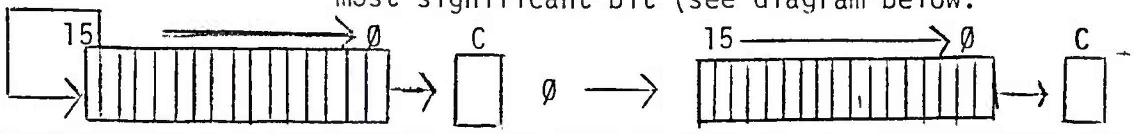
Description: The contents of D are set to zero.

CMPw	Compare
Operation:	$S_1 - S_2$
Description:	The contents of S_2 are subtracted S_1. The contents of S_1 and S_2 are unchanged. This operation sets the flags of PSW.
COMw	Complement a word
Operation:	$\overline{SD} \rightarrow SD + k$
Description:	The contents of SD are complemented when k = Ø (zero bits become one, one bits become zero). The contents of SD are negated when k = 1 (The number becomes negative by the use of a two's complement procedure.
CONT	Continue
Operation:	No operation
Description:	No operation is performed. The registers and flags are unaffected by this command.
DAJw	Decimal adjust
Operation:	$BCD\ (SD) \rightarrow SD$
Description:	The content of the word located in SD is adjusted to form a binary coded decimal (BCD) by adding a ØØ, Ø6, 6Ø, or 66 as required by each byte.
DECw	Decrement a word
Operation:	$(SD) - 1 \rightarrow SD$
Description:	The contents of SD are decremented by one.
DIV	Divide
Operation:	$SD \div S \rightarrow SD,\ Q$
Description:	The dividend (SD) is divided by the contents of S. The quotient is stored in SD. The remainder is stored in R_Q (Quotient register).
EXRw	Exclusive or with replacement
Operation:	$S \veebar R \rightarrow R$
Description:	The contents of R exclusively ORed with the contents of the address specified by S. The result is placed in R.

INCw	Increment a word
Operation:	(SD) + 1$\longrightarrow$SD
Description:	The contents of SD are incremented by one.
INij	Interrupt command
Operation:	Interrupt command (#j) of microprocessor i
Description:	Each of the 9 microprocessors included in the Master Instruction Set require specific instructions in order to provide an interrupt routine. Since they are all idiosyncratic a general command cannot adequately incorporate them all (see general comments).
JLbs	Jump and link
Operation:	Jump and link with the contents of R, conditional
Description:	JLbS, if bit b of PSW is set, transfer control to the subroutine beginning at location P store the current contents of the PC in R. JLbC, jump and link if bit b is clear.
JMbs	Jump
Operation:	Jump, conditional, unconditional
Description:	JMbs (conditional jump); test bit b of PSW. Jump to location specified by P if bit is set or clear. For example, JMbC jumps if bit b is clear and JMbS jumps if bit b is set. Use JMAS for unconditional jump.
JRbs	Test register, conditional jump
Operation:	Test register R, conditional jump
Description:	This instruction allows the contents of R to be tested. Equivalent to adding zero to the contents of R, then testing bit b of the PSW. The contents of R are unchanged but the flags of the PSW are set.
MOVw	Move a word
Operation:	(S)$\longrightarrow$D
Description:	The contents of S are transferred to D.
MPY	Multiply
Operation:	S * SD$\longrightarrow$SD, Q

Description:	The contents of S are multiplied by a second source (SD), which also stores the product. The least significant word is stored in R_Q.
OPij	Operate class
Operation:	Miscellaneous
Description:	Some commands are unique to a particular microprocessor and therefore cannot be incorporated under a general command. While these commands will of course be included in any complete MIS, they are at this time classified as an Operate Class Instruction (see specific processors for specific examples).
RESb	Reset bit b
Operation:	$\emptyset \wedge SD_b \rightarrow SD$
Description:	Bit b of word SD is cleared by logically ANDing it with zero. The result is placed in SD.
ORRw	Logical or with replacement
Operation:	$S \vee R \rightarrow R$
Description:	The contents of R are logically ORed with the contents of the address specified by S. The result is placed in R.
RIij	Return from interrupt
Operation:	Return from interrupt
Description:	Since the call to service an interrupt is unique to each microprocessor, the return from an interrupt is equally unique, and cannot be specified explicitly (see general comments).
RLnw	Rotate left
Operation:	Rotate left n times
Description:	The contents of the address specified by SD are rotated n times to the left. The carry bit can be included, (SD, C) or omitted (SD, N).
RRnw	Rotate right
Operation:	Rotate right n times
Description:	The contents of the address specified by SP are

	rotated to the carry bit can be included, (SD, C) or omitted from one rotating process (SD, N).
RTbs	Return from subroutine
Operation:	Return, conditional
Description:	RTbs; test bit b of PSW. Jump to location stored in R if tested bit is set. RTbC; return if bit b of PSW is clear. RTAS, unconditional return.
SBmw	Subtract
Operation:	$R - S\ [-C] \longrightarrow R$
Description:	The contents of S are subtracted from one contents of R. The contents of the carry bit can be included or omitted depending upon the mode, SBCw refers to binary subtraction with carry, SBNw refers to binary subtraction without carry.
SETb	Set bit b
Operation:	$1 \vee SD_b \longrightarrow SD$
Description:	The bit b of word SD is set by logically ORing it with a one. The result is placed in SD.
SLnw	Shift left
Operation:	Shift left n times
Description:	Shift the contents of location SD n times to the left. A Ø is placed in the LSB and the MSB is shifted into the carry bit.
SRnw	Shift right
Operation:	Shift right n times
Description:	The contents of the address specified by SD are rotated n times to the right. The shift can be arithmetic (SD, A) where is sign bit SD is preserved, or logical (SD, L), where a zero is put into the most significant bit (see diagram below.



XCRw	Exchange words
Operation:	$SD_1 \longleftrightarrow SD_2$

Description:	The contents of the word specified by SD_1 are "swapped" with the contents of SD_2.
XEQw	Execute contents of register R
Operation:	Execute contents of register R
Description:	Execute contents of register R.

SECTION 4

MICROPROCESSOR 1/8080/8085

4.1 INTRODUCTION

The 8080/8085 is byte oriented, having a word length of 8 bits. It has three addressing modes and a 16 bit stack pointer. It also contains six working registers which can be handled individually, or as register pairs. The 8 bit accumulator accumulates the results of arithmetic and logical operations, and there is also an 8 bit flag register. Input/output (I/O) control is achieved via I/O ports rather than regular memory space. The 8080/8085 utilizes direct control of I/O ports.

4.2 PROGRAMMING MODEL

Registers	Designation (MIS)
A	R_0
BC	R_1
DE	R_2
HL	R_3
Program Counter	R_{16}
Flag Register	R_{17}
Carry	bit Ø
Parity	bit 4
Auxiliary Carry	bit 5
Zero	bit 2
Sign	bit 3
Interrupt Mask	R_{18}
Stack Pointer	R_{19}

4.3 INTERRUPT STRUCTURE

The 8080 has a vectored type of interrupt structure with one level of interrupt. The 8085 is also vectored, with three levels of maskable interrupt and one non-maskable level.

4.4 EXAMPLES OF ADDRESSING MODES

a. Register

```
ADC   r*              ADCB              R/UR, R0
```

b. Register Indirect

```
ADC   m**             ADCB              (R3), R0
```

c. Immediate

```
ADC   i***            ADCB              #I, R0
```

d. Conditional			
CALL	JLAS	$(R_{19})-$	P
CZ	JL2S	$(R_{19})-$	P
CNZ	JL2C	$(R_{19})-$	P
CC	JLØS	$(R_{19})-$	P
CNC	JLØC	$(R_{19})-$	P
CP	JL3C	$(R_{19})-$	P
CM	JL3S	$(R_{19})-$	P
CPE	JL4S	$(R_{19})-$	P
CPO	JL4C	$(R_{19})-$	P
e. Double Register			
INX B	INCW		R_1
INX D	INCW		R_2
INX H	INCW		R_3
INX SP	INCW		R_{19}
f. Absolute			
LDA	MOVB		M, R_0
g. Implied			
CMA	COMB		R_0, $R_Ø$

* r = A, B, C, D, E, H, L

** m = memory location contained in HL

*** i = immediate

4.5 TRANSLATION INTO MIS

Table 4.5.2 presents a list of the 8080/8085 instruction set and the associated MIS translations. In order to facilitate specification of the operand(s), Table 4.5.1 has been devised.

TABLE 4.5.1

KEY TO OPERAND(S) (OP)

Op 1	Register Register Indirect Immediate
Op 2	Conditional

ASSOCIATES, INC.

Op 3	Implied
Op 4	Register Register Indirect
Op 5	Immediate
Op 6	Register
Op 7	Absolute

TABLE 4.5.2

TRANSLATION INTO MIS - 8080/8085

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
ADC	Op 1	ADCB	S	R_0
ADD	Op 1	ADNB	S	R_0
ANA	Op 1	ANRB	S	R_0
CALL	Op 2	JLAS	(R_{19})-	P
C--	Op 2	JLbs	(R_{19})-	P
CMA	Op 3	COMB	R_0	Ø
CMC	Op 3	EXRB	#0001	R_{17}
CMP	Op 1	CMPB	R_0	R
DAA	Op 3	DAJB	R_0	
DAD	Op 6	ADCW	R	R_3
DCR	Op 4	DECB	S	
DCX	Op 6	DECW	R	
DI	Op 5	SETØ	R_{18}	
EI	Op 5	RESØ	R_{18}	
HLT	Op 5	IN1Ø		
IN	Op 5	MOVB	M_p****	R_0
INR	Op 4	INCB	S	
INX	Op 6	INCW	R	
JMP	Op 2	JMAS	P	
J--	Op 2	JMbs	P	
LDA	Op 7	MOVB	M	R_0
LDAX	Op 6	MOVB	(R)	R_0
LHLD	Op 6	MOVB	M	R_3

RJB ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
LXI	Op 6	MOVW	#I	R
MOV	Op 1	MOVB	R/UR	(R_3)
MOV	Op 1	MOVB	S	R/UR
MVI	Op 5	MOVB	#I	(R_3)
MVI	Op 5	MOVB	#I	R/UR
NOP	Op 3	CONT		
ORA	Op 1	ORRB	S	R_0
OUT	Op 3	MOVB	$R_Ø$	M_p
PCHL	Op 3	MOVW	R_3	R_{16}
POP	Op 6	OP1Ø, 11		
PUSH	Op 6	OP12, 13		
RAL	Op 3	RL1B	R_0	C
RAR	Op 3	RR1B	R_0	C
R--	Op 2	RTbs	$(R_{19})+$	
RLC	Op 2	RL1B	R_0	C
RRC	Op 3	RR1B	R_0	C
RST		IN11		
SBB	Op 1	SBCB	R_0	S
SUB	Op 1	SBNB	R_0	S
SHLD	Op 7	MOVW	R_3	M
SPHL	Op 3	MOVW	R_3	R_{19}
STA	Op 7	MOVB	$R_Ø$	M
STAX	Op 6	MOVB	$R_Ø$	(R)
STC	Op 3	SETØ	R_{17}	
XCHG	Op 3	XCRW	R_2	R_3
XRA	Op 1	EXRB	(R_3)	R_0
XTHL	Op 3	XCRW	(R_{19})	R_3
		8085 ONLY		
SET		SETb	R_{18}	
CLR		RESb	R_{18}	

****M_p = memory location specifying I/O port

4.6 OP CLASS INSTRUCTIONS

OP class instructions irclude those commands that are unique to one or two of the eight microprocessors contained in the MIS. As explained in the Introduction those commands will be macro-assembled when possible. The 8080/8085 contains two such commands; POP and PUSH.

Mnemonic	MIS Mnemonic	Macro Coding		
POP rp	OP1Ø	MOVW	$(R_{19})+$	R
POP PSW	OP11	MOVW	$(R_{19})+$	R_{25}
		MOVB	UR_{25}	R_{17}
		MOVB	R_{25}	R_0
PUSH	OP12	MOVW	R	$-(R_{19})$
PUSH rp	OP13	MOVB	R_0	R_{25}
		MOVB	R_{17}	UR_{25}
		MOVW	R_{25}	$-(R_{19})$

4.7 IN/RI CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Description
HLT	IN1Ø	The processor is stopped. Registers and flags are unaffected.
RST	IN11	Restart

RJB ASSOCIATES, INC.

SECTION 5

MICROPROCESSOR 2/6800

5.1 INTRODUCTION

The 6800 microprocessor is a byte oriented processor with two general purpose registers, a Stack Pointer, two interrupt levels and six addressing modes. Negative numbers are processed by two's complement arithmetic. Those instructions requiring macro-assembly are described in Section 5.6. I/O control is memory mapped in the 6800.

5.2 PROGRAMMING MODEL

Registers	Designation (MIS)
Accumulator A	R_0
Accumulator B	R_1
Index Register X	R_2
Program Counter	R_{16}
Processor Status Register	R_{17}
Carry	bit Ø
Overflow	bit 1
Zero	bit 2
Negative	bit 3
Interrupt Mask Bit	bit Ø, R_{18}
Half-carry	bit 5
Stack Pointer	R_{19}

5.3 INTERRUPT STRUCTURE

The 6800 employs a vector fetch type of interrupt structure with two levels of interrupt request; maskable and non-maskable.

5.4 EXAMPLES OF ADDRESSING MODES

a. Immediate

ADC	ADCB	#I, R

b. Absolute

AND	ANRB	M, R

c. Relative

BRA	JMAS	(R_{16})+Off
BMI	JM3S	(R_{16})+Off
BNE	JM2C	(R_{16})+Off

BPL	JM3C	(R_{16})+Off
BVC	JM1C	(R_{16})+Off
BVS	JM2S	(R_{16})+Off
BCC	JMØC	(R_{16})+Off
BCS	JMØS	(R_{16})+Off
BEQ	JM2S	(R_{16})+Off

d. Zero Page

AND	ANRB	M, R

e. Z, Page, Indexed

AND	ANRB	M@R, R

f. Implied

ABA	ADNC	R_1, R_0

5.5 TRANSLATION INTO MIS

Table 5.5.2 presents an alphabetical listing of the 6800 instruction set along with the associated MIS translation. In order to facilitate the specification of the operand(s), Table 5.5.1 has been included.

TABLE 5.5.1

KEY TO OPERAND(S) (OP)

Op 1	Implied
Op 2	Immediate Absolute Zero Page 2, Page, X
Op 3	Absolute 2, Page, X Implied
Op 4	Relative
Op 5	Absolute 2, Page, X
Op 6	Absolute Zero Page 2, Page, X

TABLE 5.5.2

TRANSLATION INTO MIS - 6800

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
ABA	Op 1	ADNB	R_1	R_0
ADC	Op 2	ADCB	M	R
ADD	Op 2	ADNB	M	R
AND	Op 2	ANRB	M	R
ASL	Op 3	SL1B	SD	
ASR	Op 3	SR1B	SD	A
BCC	Op 4	JMØC	(R_{16})+Off	
BCS	Op 4	JMØS	(R_{16})+Off	
BEQ	Op 4	JM2S	(R_{16})+Off	
BGE	Op 4	JM7S	(R_{16})+Off	
BGT	Op 4	JM6S	(R_{16})+Off	
BHI	Op 4	JM8S	(R_{16})+Off	
BIT	Op 2	ANDB	R_0/R_1	M
BLE	Op 4	JM6C	(R_{16})+Off	
BLS	Op 4	JM8C	(R_{16})+Off	
BLT	Op 4	JM7C	(R_{16})+Off	
BMI	Op 4	JM3S	(R_{16})+Off	
BNE	Op 4	JM2C	(R_{16})+Off	
BPL	Op 4	JM3C	(R_{16})+Off	
BRA	Op 4	JMAS	(R_{16})+Off	
BSR	Op 4	JLAS	(R_{19})-	(R_{16})+Off
BVC	Op 4	JM1C	(R_{16})+Off	
BVS	Op 4	JM1S	(R_{16})+Off	
CBA	Op 1	CMPB	R_0	R_1
CLC	Op 1	RESØ	R_{17}	
CLI	Op 1	RESØ	R_{18}	
CLR	Op 3	CLRB	SD	
CLV	Op 2	RES1	R_{17}	
CMP	Op 3	CMPB	R_0/R_1	M
COM	Op 3	COMB	SD	Ø
CPX	Op 2	CMPB	R_2	M

ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
DAA	Op 1	DAJB	SD	
DEC	Op 3	DECB	SD	
DES	Op 1	DECW	R_{19}	
DEX	Op 1	DECW	R_2	
EOR	Op 2	EXRB	M	R_0/R_1
INC	Op 3	INCB	SD	R_0/R_1
INS	Op 1	INCW	R_{19}	
INX	Op 1	INCW	R_2	
JMP	Op 5	JMAS	P	
JSR	Op 5	JLAS	$(R_{19})-$	P
LDA	Op 2	MOVW	M	R_0
LDS	Op 2	MOVW	M	R_{19}
LDX	Op 2	MOVW	M	R_2
LSR	Op 3	SR1B	SD	L
NEG	Op 3	COMB	SD	1
NOP	Op 1	CONT		
ORA	Op 2	ORRB	S	R_0/R_1
PSH	Op 1	MOVB	R	$(R_{19})-$
PUL	Op 1	MOVB	$+(R_{19})$	R_0/R_1
ROL	Op 3	RL1B	SD	C
ROR	Op 3	RR1B	SD	C
RTI	Op 1	RI2Ø		
RTS	Op 1	RTAS	$+(R_{19})$	
SBA	Op 1	SBNB	R_0	R_1
SBC	Op 2	SBCB	R_0/R_1	M
SEC	Op 1	SETØ	R_{17}	
SEI	Op 1	SETØ	R_{18}	
SEV	Op 1	SET1	R_{17}	
STA	Op 6	MOVB	R_0/R_1	M
STS	Op 6	MOVW	R_{19}	M
STX	Op 6	MOVW	R_2	M
SUB	Op 2	SBNB	R_0/R_1	M
SWI	Op 1	IN2Ø		

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
TAB	Op 1	MOVB	R_0	R_1
TAP	Op 1	MOVB	R_1	R_{17}
TBA	Op 1	MOVB	R_1	R_0
TPA	Op 1	MOVB	R_{17}	R_0
TST	Op 3	SLØB	SD	
TSX	Op 1	MOVW	R_{19}	R_2
TXS	Op 1	MOVW	R_2	R_{19}
WAI	Op 1	IN21		

5.6 OP CLASS INSTRUCTIONS

The 6800 does not contain any commands which require macro-assembly by the MIS.

5.7 IN/RI CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Description
SWI	IN2Ø	Software interrupt
WAI	IN21	Wait for interrupt
RTI	RI2Ø	Return from interrupt

SECTION 6

MICROPROCESSOR 3/TMS-9900

6.1 INTRODUCTION

The TMS-9900 has a 16 bit word length, a 32-kword address space and a set of 69 instructions. The internal architecture of the 9900 allows for 16 general purpose registers and 15 index registers. A unique feature of this microprocessor is its "workspace register file" capability. This file occupies 16 contiguous memory words in the general memory area. The workspace register points to the first of the general purpose registers set up in the RAM space. I/O control is direct in the 9900.

6.2 PROGRAMMING MODEL

Registers	Designation (MIS)
Program Counter	R_{16}
Status Register	R_{17}
Equal	bit 2
Arithmetically $>$	bit 8
Logically $>$	bit 9
$\geq$	bit 7
Arithmetically $<$	bit B
Parity	bit 4
Carry	bit Ø
Overflow	bit E
Unconditional	bit A
Interrupt Register	R_{18}
Workspace Register	R_{19}
Communications Register Unit (CRU)	R_{28}

6.3 INTERRUPT STRUCTURE

The TMS-9900 has a vector-type interrupt structure with 16 levels and each is maskable.

6.4 EXAMPLES OF ADDRESSING MODELS

a. Immediate

LWPI	MOVW	#I, R_{19}
LI	MOVW	#I, (R_{19})+Off
LIMI	MOVW	#I, R_{18}
LDCR	MOVW	#I, R_{28}

ASSOCIATES, INC.

b. Workspace Register

CLR r	CLRW	(R_{19})+Off

c. Workspace Register Indirect

CLR *r	CLRW	$(R_{19}$+Off)

d. Register Indirect With Autoincrement

CLR *r+	CLRW	$(R_{19}$+Off)+

e. Indexed

CLR @TABLE(r)	CLRW	M@$(R_{19}$+Off)

f. Program Counter Relative

JEQ	JM2S	P
JGT	JM8S	P
JH	JM6S	P
JHE	JM7S	P
JL	JM7C	P
JLE	JM6C	P
JLT	JMDS	P
JOP	JM4C	P
JOC	JMØS	P
JNE	JM1C	P
JNO	JM2C	P
JNC	JMØC	P
JMP	JMAS	P

g. Direct

CLR @m	CLRW	M

h. Communications Register Unit Relative

SBO	SETb	(R_{28})+Off

6.5 TRANSLATION INTO MIS

Table 6.5.2 lists the instruction set of the TMS-9900 and the associated translations. The operand(s) used by the 9900 have been recoded as defined in Table 6.5.1.

TABLE 6.5.1

KEY TO OPERAND(S) (OP)

Op 1 (Dual)	Workspace Register Workspace Register Indirect Direct Indexed Workspace Register Indirect Auto-increment
Op 2 (Single)	Workspace Register Workspace Register Indirect Direct Indexed Workspace Register Indirect Auto-increment
Op 3	Immediate
Op 4	Workspace Register
Op 5	CRU Relative Addressing
Op 6	Program Counter Relative Addressing
Op 7	Internal Register Store

TABLE 6.5.2

TRANSLATION INTO MIS - TMS-9900

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
A	Op 1	ADNW	S	SD
AB	Op 1	ADNB	S	SD
ABS	Op 1	OP3Ø		
AI	Op 3	ADNW	#I	(R_{19})+0ff
ANDI	Op 3	ANRW	#I	(R_{19})+0ff
B	Op 1	JMAS	P	
BL	Op 1	JLAS	(R_{19})+11	P
BLWP	Op 1	OP31	P	
C	Op 1	CMPW	S_1	S_2
CB	Op 1	CMPB	S_1	S_2
CI	Op 3	CMPW	#I	S_2

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
CKOF		IN3Ø		
CKON		IN31		
CLR	Op 1	CLRW	SD	
COC	Op 2	OP32		
CZC	Op 2	OP33		
DEC	Op 1	DECW	SD	
DECT	Op 1	OP38		
DIV	Op 2	DIVW	S	SD
IDLE		IN32		
INC	Op 1	INCW	SD	
INCT	Op 1	OP37		
INV	Op 1	COMW	SD	Ø
JEQ	Op 6	JM2S	P	
JGT	Op 6	JM8S	P	
JH	Op 6	JM6S	P	
JHE	Op 6	JM7S	P	
JL	Op 6	JM7C	P	
JLE	Op 6	JM6C	P	
JLT	Op 6	JMDS	P	
JOP	Op 6	JM4C	P	
JOC	Op 6	JMØS	P	
JNE	Op 6	JM1C	P	
JNO	Op 6	JM2C	P	
JNC	Op 6	JMØS	P	
JMP	Op 6	JMAS	P	
LDCR	Op 1	MOVW	S	R_{28}
LI	Op 3	MOVW	#I	(R_{19})+Off
LIMI	Op 3	MOVW	#I	R_{18}
LREX		IN33		
LWPI	Op 3	MOVW	#I	R_{19}
MOV	Op 1	MOVW	S	D
MOVB	Op 1	MOVB	S	D

RJB ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
MPY	Op 2	MPYW	S	SD
NEG	Op 1	COMW	(R_{19})+Off	1
ORI	Op 3	ORRW	#I	(R_{19})+Off
RSET		IN34		
RTWP		OP34		
S	Op 1	SBNW	SD	S
SB	Op 1	SBNB	SD	S
SBO	Op 5	SETb	R_{28}	
SBZ	Op 5	RESb	R_{28}	
SETO	Op 1	MOVW	#FFFF	(R_{19})+Off
SLA	Op 4	SL1W	SD	
SOC	Op 1	ORRW	S	SD
SOCB	Op 1	ORRB	S	SD
SRA	Op 4	SR1W	(R_{19})+Off	A
SRC	Op 4	RR1W	(R_{19})+Off	N
SRL	Op 4	SR1W	(R_{19})+Off	L
STCR	Op 1	MOVW	R_{28}	(R_{19})+Off
STST	Op 7	MOVW	R_{17}	(R_{19})+Off
STWP	Op 7	MOVW	R_{19}	(R_{19})+Off
SWPB	Op 1	XCRB	SD_1	SD_2
SZC	Op 1	OP35		
SZCB	Op 1	OP36		
TB	Op 5	ANDW	S	R_{28}
X	Op 1	XEQW	(R_{19})+Off	
XOP	Op 1	IN35		
XOR	Op 2	EXRW	S	(R_{19})+Off

6.6 OP CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Macro Coding		
ABS	OP3Ø	JRFC	SD	R_{16}+SKIP*
		COMW	SD	1

Mnemonic	MIS Mnemonic	Macro Coding		
BLWP	OP31	MOVW	R_{19}	R_0
		MOVW	S	R_{19}
		MOVW	R_0	$(R_{19})+13$
		MOVW	R_{16}+SKIP	$(R_{19})+14$
		MOVW	R_{17}	$(R_{19})+15$
		JMAS	S_{+1}**	
COC	OP32	MOVW	S_1	R_0
		ANRW	S_2	R_0
		CMPW	S_2	R_0
CZC	OP33	MOVW	S_1	R_0
		COMW	R_0	Ø
		ANRW	S_2	R_0
		CMPW	S_2	R_0
RTWP	OP34	MOVW	$(R_{19})+15$	R_{17}
		MOVW	$(R_{19})+14$	R_{16}
		MOVW	$(R_{19})+13$	R_{19}
SZC	OP35	MOVW	MASK	R_0
		COMW	R_0	Ø
		ANRW	R_0	S
SZCB	OP36	MOVB	MASK	R_0
		COMB	R_0	Ø
		ANRB	R_0	S
INCT	OP37	INCW	SD	
		INCW	SD	
DECT	OP38	DECW	SD	
		DECW	SD	

*Offset required to skip past macro code..

**S_{+1} = effective address formed by S, plus one

6.7 IN/RI CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Description
CKOF	IN3Ø	Address lines A_0 - A_2 set: HHL
CKON	IN31	Address lines A_0 - A_2 set: HLH

Mnemonic	MIS Mnemonic	Description
IDLE	IN32	Suspend instruction execution until an interrupt, load or reset occurs.
LREX	IN33	Load or restart execution.
RSET	IN34	Computer reset
XOP	IN35	Extended operation

SECTION 7

MICROPROCESSOR 4 - LITTON HARS

7.1 INTRODUCTION

The Litton HARS microprocessor utilizes a 16/32 word size (instructions, 16 bits, data, 32 bits). HARS processes negative numbers by two's complement arithmetic. Unfortunately, due to insufficient documentation, the type of I/O control cannot be determined with certainty. However, the inclusion of the BMV command (Block move of memory or I/O) suggests that I/O control is the direct type.

7.2 PROGRAMMING MODEL

Registers	Designation (MIS)
Accumulator	R_0
Program Counter	R_{16}
Processor Status Word	R_{17}
Interrupt Mask	R_{18}
Stack Pointer	R_{19}
Extension Register	R_{28}

7.3 INTERRUPT STRUCTURE

Again, due to insufficient documentation on the HARS, the type of interrupt structure is not known. However, it appears to have one level of interrupt, which is maskable.

7.4 EXAMPLES OF ADDRESSING MODES

a. Register

AN	ANRW	S, R_0

b. Extended

ANE	ANRW	M@R*, R_0

c. Extended, Indirect

ANE*	ANRW	(M@R*), R_0

d. Address

ANY	ANRW	M, R_0

e. Address, Indirect

ANY*	ANRW	(M), R_0

f. Immediate

ANM	ANRW	#I, R_0

7.5 TRANSLATION INTO MIS

Table 7.5.1 lists the instruction set of the HARS microprocessor along with the associated MIS translations. Due to a lack of sufficient documentation, the operands used by HARS are unknown, hence unspecified.

TABLE 7.5.1

TRANSLATION INTO MIS- LITTON HARS

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
R		IN4Ø		
IR		RI4Ø		
RTN		RTAS	R_{19}	
ASM		ADNB	#I	$R_Ø$
SSM		SBNB	#I	$R_Ø$
SF		OP4Ø		
RF		OP41		
II		SETØ	R_{18}	
CR		CMPW	S_1	S_2
LR		MOVW	S	R
EX		XCRW	SD_1	SD_2
AR		ADNW	R	R_0
SR		SBNW	R_0	R
DN		*		
DIS		*		
SLL		SL1W	SD	
SRL		SR1W	SD	L
SRA		SR1W	SD	A
SLD		SL1D	SD	
SRD		SR1D	SD	A
NOT		COMW	SD	Ø
NEG		COMW	SD	1
BMV		OP42		
BT		ANDW	S	R
L		MOVW	M	R_0
DL		MOVD	M	R_0

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
ST		MOVW	R_0	M
DT		MOVD	R_0	M
AN		ANRW	S	R_0
OR		ORRW	S	R_0
T		JMAS	P	
TP		JM7S	P	
TM		JM7C	P	
TZ		JM2S	P	
DAC		ADCD	R	R_0
DEX		XCRD	R_0	R_2
A		ADNW	S	R_0
S		SBNW	R_0	S
M		MPYW	S	SD
D		DIVW	S	SD
DA		ADND	S	SD
DS		SBND	SD	S
TR		OP4j	R	
TI		JLAS	R_{19}	(M)

* Insufficient documentation to determine the meaning of the command.

7.6 OP CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Macro Coding	
SF	OP4Ø	Save register file	
RF	OP41	Restore register file	
BMV	OP42	Block move of memory or I/O	
TR	OP4j**	INCW JM3S	R P

** j = 3 - 7, where the numbers 3 - 7 refer to addressing modes a - e listed in Section 7.4.

7.7 IN/RL CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Description
R	IN4Ø	Restart
IR	RI4Ø	Return from interrupt

SECTION 8

MICROPROCESSOR 5 - SDP 175

8.1 INTRODUCTION

The SDP 175 microprocessor utilizes a 16 bit word. Due to sparse documentation, other characteristics of the SDP 175 are not known. It is assumed that there are 16 general purpose registers, and two index registers. I/O control is memory mapped in the SDP 175.

8.2 PROGRAMMING MODEL

Registers	Designation (MIS)
A Register	R_0
B Register	R_1
Program Counter	R_{16}
Processor Status Register	R_{17}
Overflow	bit 1
Zero	bit 2
Negative	bit F
$>$	bit 6
$\geq$	bit 7
Interrupt Mask	R_{18}
Index Register, X	R_{20}
Index Register, Y	R_{21}
Error Register	R_{22}

8.3 INTERRUPT STRUCTURE

The type of interrupt structure utilized by the SDP 175 cannot be determined due to insufficient information.

8.4 EXAMPLES OF ADDRESSING MODE

Examples of addressing modes cannot be provided, due to insufficient information.

8.5 TRANSLATION INTO MIS

Table 8.5.1 presents the list of the SDP 175 instruction set along with the associated MIS translations. Due to insufficient information the operands used by the SDP 175 are not known.

TABLE 8.5.1

TRANSLATION INTO MIS - SDP 175

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
<u>Memory Ref With Indexing</u>				
LDR		MOVW		
STR		MOVW		
CMR		CMPW		
ADDM		ADNW		
SUBM		SBNW		
MPYM		MPYW		
ANDM		ANRW		
ORM		ORRW		
INCM		INCW		
LDRD		MOVD		
STRD		MOVD		
ADDB		ADND		
SUBD		SBND		
<u>Memory Reference-Pre-Indexed Indirect</u>				
LRI		MOVW	$(M@R_{20})$	R
SRI		MOVW	R	$(M@R_{20})$
LRIX		MOVW	$(M@R_{20})+$	R
SRIX		MOVW	$(M@R_{20})+$	R
<u>Unconditional Jump</u>				
JMP		JMAS	P	
JSR		JLAS	$(R_{19})+$	P
JMPI		JMAS	(P)	
JSRI		JLAS	$(R_{19})+$	(P)
<u>Conditional Jump</u>				
JZ		JM2S	P	
JN		JM2C	P	
JP		JMFC	P	
JM		JMFS	P	

ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
JOT		JMES	P	
JLE		JM6C	P	
JGE		JM7S	P	
JLT		JM7C	P	
JGT		JM6S	P	
IJX		OP5C		
IJY		OP5D		
<u>Register/Register</u>				
RTR		MOVW	R	R
ADD		ADNW	R	R
SUB		SBNW	R	R
MPY		MPYW	R	R
AND		ANRW	R	R
OR		ORRW	R	R
XOR		EXRW	R	R
ADDL		OP5Ø		
SUBL		OP51		
SWAP		XCRW	R	R
SUBC		CMPW	R	R
ANDC		ANDW	R	R
XORC		OP52		
DIV		DIVD	R	R
<u>BIT</u>				
TBIT		ANDW	#MASK	R
SBIT		SETb	R	
RBIT		RESb	R	
TBITI		ANDW	(M)	R
SBITI		SETb	(R)	
RBITI		RESb	(R)	
<u>Register Operate</u>				
ABS		OP53		
INCR		INCW	R	
DECR		DECW	R	
CMPL		COMW	R	Ø

ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
NEG		COMW	R	1
ZERO		CLRW	R	
ZLBY		CLRB	UR	
ZRBY		CLRB	R	
XEC		XEQW	R	
LMDT		OP54		
RTRN		RTAS	R	
<u>Literal</u>				
LDVS		MOVB	#I	R
ADDVS		ADNB	#I	R
LDV		MOVW	#I	R
ADDV		ADNW	#I	R
ANDV		ANRW	#I	R
ORV		ORRW	#I	R
OXRV		EXRW	#I	R
SUBVC		CMPW	#I	R
ANDVC		ANDW	#I	R
XORVC		OP55		
<u>Shift</u>				
SRA	R, n	SRnW	R	A
LSRA	n	SRnD	R_0	A
SRC	R, n	RRnW	R	N
SLSF	n	OP56		
SL1M	n	OP57		
SLA	R, n	SLnW	R	
LSLA	n	SLnD	R_0	
<u>Control</u>				
INE		RESØ	R_{18}	
INH		SETØ	R_{18}	
WAIT	d	IN5Ø		
NOP		CONT		
LMSK	R	MOVB	R	R_{18}
CPLSE		OP58		

RJB ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
Stack Operations				
PUSH		MOVW	R	$(R_{19})+$
PULL		MOVW	$-(R_{19})$	R
TSKR		MOVW	R_{19}	R
TRSK		MOVW	R	R_{19}
TMSK	a	MOVW	M	R_{19}
PUSHS		MOVW	R_{17}	$(R_{19})+$
PULLS		MOVW	$-(R_{19})$	R_{17}
RTRNS		RTAS	$-(R_{19})$	
Test Operations				
TPLSE		OP59		
CKSUM		OP5A		
FBIT	K, R	SETb	R	
FBIT	K, R	SETb	(R)	
RESET		IN51		
ERIN	R	MOVW	R_{22}	R
DIN	R	MOVW	M_p	R
OUT	K, R	OP5B		

8.6 OP CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Macro Coding		
ADDL	OP5Ø	Add and limit overflow		
SUBL	OP51	Subtract and limit overflow		
XORC	OP52	MOVW EXRW	S_1 S_2	R_{23} R_{23}
ABS	OP53	JRFC COMW	R R	(R_{16})+SKIP* 1
LMOT	OP54	Limit if overflow true		
XORVC	OP55	MOVW EXRW	#I R	R_{23} R_{23}
SLSF	OP56	Shift left scale factor		
SLIM	OP57	Scaled limit		
CPLSE	OP58	Output control pulse		

Mnemonic	MIS Mnemonic	Macro Coding		
TPLSE	OP59	Test pulse		
CKSUM	OP5A	Check sum		
OUT K, R	OP5B	There is insufficient information to describe this I/O command.		
IJX	OP5C	JR6C	R_{20}	(R_{19})+SKIP
		DECW	R_{20}	
		JMAS	P^{20}	
IJY	OP5D	JR6C	R_{21}	(R_{19})+SKIP
		DECW	R_{21}	
		JMAS	P^{21}	

* SKIP = Offset required to skip past macro coding.

8.7 IN/RI CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Description
WAIT d	IN5Ø	Wait for interrupt
RESET	IN51	Reset

SECTION 9

MICROPROCESSOR 6 - MECA-43

9.1 INTRODUCTION

The MECA-43 is a word-oriented processor (16 bits) with 16 general registers and six addressing modes. It processes negative numbers by two's complement arithmetic and has a 32 or 48 bit floating point. The MECA-43 employs 16 vectored interrupts that are maskable and expandable. I/O control in the MECA-43 is direct.

9.2 PROGRAMMING MODEL

Registers		Designation (MIS)	
Processors 1	Processor 2	Processor 1	Processor 2
Accumulator (A)	Accumulator (A)	R_0	R_8
Quotient (B)	Quotient (B)	R_1	R_9
MR	MR	R_2	R_{10}
Base/Return (BR)	Base/Return (BR)	R_3	R_{11}
MSP	MSP	R_4	R_{12}
X	X	R_5	R_{13}
Y	Y	R_6	R_{14}
Z	Z	R_7	R_{15}
Program Counter		R_{16}	
Processor Status Word (PSW)		R_{17}	
Interrupt Mask (IM)		R_{18}	
Alternate Program Counter		R_{19}	
Extension Register		R_{28}	R_{29}

9.3 INTERRUPT STRUCTURE

The MECA-43 employs 16 vectored interrupts that are maskable and expandable.

9.4 EXAMPLES OF ADDRESSING MODES

a. Common

LDA	MOVW	M, R_0/R_8

b. Direct

LDA	MOVW	M@R_3/R_{11}

c. Indirect

LDA	MOVW	(M@R_3/R_{11})

RJB ASSOCIATES, INC.

d. Immediate		
LDA	MOVW	#I, R_0/R_8
e. Program Counter Relative		
LDA	MOVW	(R_{16})+Off, R_0/R_8
f. Index Register Relative		
LDA	MOVW	(R_5/R_{13})+Off, R_0/R_8
LAXi*	MOVW	(R_{i+4}/R_{12+i})+Off, R_0/R_8

* i = 1, 2, 3

9.5 TRANSLATION INTO MIS

The instruction set of the MECA-43 is listed in Table 9.5.2 along with the associated MIS translations. The operands utilized by the MECA-43 are specified according to Table 9.5.1.

TABLE 9.5.1

KEY TO OPERAND(S) (OP)

Op 1	Common Indirect Direct Immediate (P) Relative (X) Relative
Op 2	Inter-register
Op 3	Immediate
Op 4	Direct
Op 5	Common
Op 6	Immediate (X) Relative
Op 7	Common Indirect Direct (P) Relative (X) Relative
Op 8	Indirect
Op 9	P-relative
Op 10	(X) Relative

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
<u>Data Transfer (A)</u>				
LDA	Op 1	MOVW	S	R_0/R_8
LAXi	Op 2	MOVW	R	R_0/R_8
LAFB	Op 2	MOVW	R_1	R_0/R_8
LABR	Op 2	MOVW	R_3	R_0/R_8
IBAR	Op 3	MOVB	#I	R_0/R_8
EXBA	Op 2	XCRB	R_0/R_8	UR_0/UR_8
STA	Op 1	MOVW	R_0	D
<u>Data Transfer (B)</u>				
LDB	Op 1	MOVW	S	R_1/R_9
LBFA	Op 2	MOVW	R_0	R_1/R_9
IBBR	Op 3	MOVB	#I	R_1/R_9
EXBB	Op 2	XCRB	R_1/R_9	UR_1/UR_9
STB	Op 1	MOVW	R_1/R_9	D
<u>Data Transfer (X)</u>				
LXD	Op 4	MOVW	M	R_5
LXI	Op 3	MOVW	#I	R_5
LXiA	Op 2	MOVW	R_{i+4}	R_0
SXC	Op 5	MOVW	R_5	M
SXD	Op 4	MOVW	R_5	M
<u>Data Transfer (BR)</u>				
LBRD	Op 4	MOVW	M	R_3
LBRI	Op 3	MOVW	#I	R_3
LBRA	Op 2	MOVW	R_0	R_3
IBBA	Op 3	MOVB	#I	R_3
SBRC	Op 5	MOVW	R_3	M
SBRD	Op 4	MOVW	R_3	M
<u>Add (A)</u>				
ADD	Op 1	ADCW	S	R_0/R_8
ADBA	Op 2	ADCW	R_1	R_0/R_8
AXiA	Op 2	ADCW	R_{i+4}	R_0/R_8
<u>Add (B)</u>				
ADB	Op 1	ADCW	S	R_1/R_9
ADAB	Op 2	ADCW	R_0	R_1/R_9

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
Add (X)				
AXD	Op 4	ADCW	M	R_5
AXI	Op 6	ADCW	#I	R_5
Add (BR)				
ABRD	Op 4	ADCW	M	R_3
ABRI	Op 3	ADCW	#I	R_3
Subtract (A)				
SUB	Op 1	SBCW	R_0/R_8	S
RSA	Op 1	OP6Ø		
SBFA	Op 2	SBCW	R_0/R_8	R_1/R_9
Subtract (B)				
SBB	Op 1	SBCW	R_1/R_9	S
Multiply/Divide				
MPY	Op 1	MPYW	S	R_0/R_8
DIV	Op 1	DIVW	S	R_0/R_8
Logical (A)				
AND	Op 1	ANRW	S	R_0/R_8
LDR	Op 1	ORRW	S	R_0/R_8
XDR	Op 1	EXRW	S	R_0/R_8
Shift (A)				
SARn	Op 6	SRnW	R_0/R_8	A
SALn	Op 6	SLnW	R_0/R_8	
Shift (B)				
SBR	Op 6	SRnW	R_1/R_9	A
SBL	Op 6	SLnW	R_1/R_9	
Shift (A) and (B)				
SDR	Op 6	SR1D	R_0/R_8	A
SDL	Op 6	SL1D	R_0/R_8	
NRM	Op 6	OP61		
Transfer Unconditional				
TRA	Op 1	JMAS	P	
TRS	Op 7	JLAS	$(R_3)+$	P
TTR	Op 8	RTAS	$-(R_3)$	

ASSOCIATES, INC.

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
<u>Transfer (A)</u>				
TRZ	Op 1	JR2S	R_0/R_8	P
TRN	Op 1	JR3S	R_0/R_8	P
<u>Transfer (X)</u>				
TXI	Op 1	JRBC	$-R_{i+4}$	(R_{i+4})
TXP	Op 9	JRBC	$-R_{i+4}$	(R_{16})+Off
<u>Transfer (BR)</u>				
TBRI	Op 8	JRBC	$-R_3$	(R_3)
TBRP	Op 9	JRBC	$-R_3$	(R_{16})+Off
<u>Skip</u>				
SMP	Op 1	CMPW	S	R_0
DSØ	Op 6	JR2S	M_p	(R_{16})+Off
DS1	Op 6	JR2C	M_p	(R_{16})+Off
<u>Input/Output</u>				
INA	Op 6	MOVW	M_p	R_0/R_8
INB	Op 6	MOVW	M_p	R_1/R_9
OTA	Op 6	MOVW	R_0/R_8	M_p
OTB	Op 6	MOVW	R_1/R_9	M_p
<u>Double Precision</u>				
DAD	Op 1	ADCD	S	R_0/R_8
DSU	Op 1	SBCD	R_0/R_8	S
DLD	Op 1	MOVD	M	R
DST	Op 1	MOVD	R	M
DML	Op 1	MPYD	S	R_0/R_8
DDV	Op 1	DIVD	S	R_0/R_8
<u>Floating Point</u>				
FLD	Op 10	OP62		
FXD	Op 10	OP63		
FAD	Op 1	OP64		
FSD	Op 1	OP65		
FMD	Op 1	OP66		
FDO	OP 1	OP67		
FLS	Op 10	OP68		
FXS	Op 10	OP69		

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
FAS	Op 1	OP6A		
FSS	Op 1	OP6B		
FMS	Op 1	OP6C		
FDS	Op 1	OP6D		

9.6 OP CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Macro Coding		
RSA	OP6∅	MOVW SBCW MOVW	S_1 R_{20} R_{20}	R_{20} R_0/R_8 R_0/R_8
NRM	OP61	See MECA manual Section 3.6.		
FLD	OP62	Fix to float conversion. See MECA manual Section 3.7.3.		
FXD	OP63	Fix to float conversion. See MECA manual Section 3.7.1.		
FAD	OP64	Floating point add, double precision. See MECA manual Section 3.7.5.		
FSD	OP65	Floating point subtract, double precision. See MECA manual Section 3.7.7.		
FMD	OP66	Floating point multiply, double precision. See MECA manual Section 3.7.9.		
FDD	OP67	Floating point divide, double precision. See MECA manual Section 3.7.11.		
FLS	OP68	Fix to float conversion. See MECA manual Section 3.7.4.		
FXS	OP69	Fix to float conversion. See MECA manual section 3.7.2.		
FAS	OP6A	Floating point add, single precision. See MECA manual Section 3.7.6.		
FSS	OP6B	Floating point subtract, single precision. See MECA manual Section 3.7.8.		
FMS	OP6C	Floating point multiply, single precision. See MECA manual Section 3.7.10.		
FDS	OP6D	Floating point divide, single precision. See MECA manual Section 3.7.12.		

9.7 IN/RI CLASS INSTRUCTIONS

The MECA-43 does not contain any IN/RI instructions.

9.8 FINAL NOTE

As seen from Section 9.2, the MECA contains two sets of general purpose registers. One set (designated Processor 1) is assigned to the Executive Processor, intended for general processing, whereas the second set (designated Processor 2) is assigned to the I/O Processor. Since the MECA-43 is described as 2 processors, a second PC (R_{19} - Alternate Program Counter) was included. It is possible that a second PSW may be required to facilitate correct switch-back to the alternate processor.

SECTION 10

MICROPROCESSOR 7 - SYMGEN

10.1 INTRODUCTION

The SYMGEN (Symbol Generator) microprocessor is a 16 bit machine with 16 general purpose registers. It processes negative numbers via two's complement arithmetic. The SYMGEN has a vectored interrupt structure with two high priority non-maskable interrupts and four maskable interrupts. I/O control in the SYMGEN is of the direct type.

10.2 PROGRAMMING MODEL

Registers	Designation (MIS)
R_x, x = 0, 1, 2, ...15	R_0-R_{15}
R_y, y = 0, 1, 2, ...15	R_0-R_{15}
Program Counter	R_{16}
Processor Status Word	R_{17}
Zero	bit 2
Negative	bit F
$\geq$	bit 7
Interrupt Register	R_{18}
Stack Pointer	R_{19}
General Purpose Register	R_{21}
External Status Register	R_{28}

10.3 INTERRUPT STRUCTURE

The SYMGEN employs a vectored interrupt system with two high-priority non-maskable interrupts and four maskable interrupts.

10.4 EXAMPLES OF ADDRESSING MODES

a. Register

$R_y = R_x$	MOVW	R_x, R_y

b. Storage Reference

$R_xB = C$	MOVB	#I, R_x
$R_x = C$	MOVW	S, R_x

c. Indirect Via Registers

LDI R_y @ R_x	MOVW	(R_x), R_y

d. Indirect Via Registers, Pre-increment

LDI R_y @ R_x + 1	MOVW	$+(R_x)$, R_y

e. Offset, Indirect Via Register

RLD	R_x @ Displacement	MOVW	$(R_{16}$+Off), R_x

f. INDIRECT

LDI	R_y @ R_x	MOVW	(R_x), R_y

g. Indexed Absolute

LDD	R_y @ R_x + Address	MOVW	M@R_x, R_y

h. Conditional

JMP	Address	JMAS	P
JSR	Address	JLAS	R_o, P
JPR	R_x	JMAS	(R_x)
JPR	R_x + 1	JMAS	+(R_x)
JIF	R_x, Neg Address	JRFS	R_x, (R_{16})+Off
JAD	R_x, Address	JR7S	R_x-, P
JAD	R_y, R_x	JR7S	R_y-, (R_x)
DJSR	Address	JLAS	R_0, P
DJSR	R_x, Address	JLAS	R_x, P
IF R_x.GE. R_y SKIP		OP7B	

10.5 TRANSLATION INTO MIS

TABLE 10.5.1

TRANSLATION INTO MIS - SYMGEN

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
<u>Register to Register</u>				
$R_y = R_x$		MOVW	R_x	R_y
$R_y = -R_x$		OP7Ø		
$R_y = R_xF$		OP71		
$R_y = R_x + R_y$		ADNW	R_x	R_y
$R_y = R_y - R_x$		SBNW	R_y	R_x
$R_y = R_x - R_y$		OP72		
$R_y = R_x$ AND R_y		ANRW	R_x	R_y
$R_y = R_x$ OR R_y		ORRW	R_x	R_y
$R_y = R_x$ XOR R_y		EXRW	R_x	R_y
$R_y = R_x * R_y$		MPYW	R_x	R_y
$R_y, R_x = R_y \backslash R_x$		OP73		

Mnemonic Operand(s)	MIS Mnemonic	Operand 1	Operand 2
R_y, R_x = $R_y \backslash\backslash R_x$	OP74		
R_y = -- R_x	OP75		
R_y = R_y ++ R_x	ADND	R_x	R_y
R_y = R_y -- R_x	SBND	R_y	R_x
Q = R_x	MOVW	R_x	R_{30}
R_y = Q	MOVW	R_{30}	R_y
<u>Register Operate</u>			
EXEC R_x	XEQW	R_x	
R_x(N) = 1	SETb	R_x	
R_x(N) = 0	RESb	R_x	
<u>Register Shift</u>			
SHL R_x, COUNT	SLnW	R_x	
SHR R_x, COUNT	SRnW	R_x	L
SHR R_x, COUNT, MSB	SRnW	R_x	A
RTL R_x, COUNT	RLnW	R_x	N
RTR R_x, COUNT	RRnW	R_x	N
DSHL R_x, COUNT	SLnD	R_x	
DSHR R_x, COUNT	SRnD	R_x	L
DSHR R_x, COUNT, MSB	SRnD	R_x	A
<u>Storage Reference</u>			
R_x = C	MOVW	S	R_x
R_y = R_x + C	OP76		
R_y = R_x AND C	OP77		
R_y = R_x OR C	OP78		
LDD R_x @ C	MOVW	S	R_x
STD R_x @ C	MOVW	R_x	S
LDD R_y @ R_x + C	MOVW	M@R_x	R_y
STD R_y @ R_x + C	MOVW	R_y	M@R_x
LDI R_y @ R_x	MOVW	(R_x)	R_y
STI R_x @ R_y	MOVW	R_x	(R_y)
R_xB = C	MOVB	#I	R_x
R_xB = R_x + C	ADNB	#I	R_x
R_xB = R_x - C	SBND	R_x	#I
RLD R_x @ DISPLACEMENT	MOVW	(R_{16})+Off	R_x
RST R_x @ DISPLACEMENT	MOVW	R_x	(R_{16})+Off

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
LDI R_y @ R_x + 1		MOVW	$+(R_x)$	R_y
$R_y = R_x - C$		OP79		
<u>Stack Operations</u>				
PUSH		MOVW	R_x	$+(R_{19})$
PUSH R_x @ R_y		MOVW	R_x	$+(R_y)$
STI R_x @ R_y + 1		MOVW	R_x	$+(R_y)$
POP R_y		MOVW	$(R_{19})-$	R_y
POP R_y @ R_x		MOVW	$(R_x)-$	R_y
<u>Storage To Storage</u>				
MTRAN Q WORDS		OP7A		
FROM R_x TO R_y				
<u>Unconditional Transfers</u>				
JMP ADDRESS		JMAS	P	
JSR ADDRESS		JLAS	R_0	(R_{16})+Off
JSR ADDRESS		JLAS	R_0	M
RETURN		RTAS	$(R_{19})-$	
RETURN R_x		RTAS	$(R_x)-$	
IRET		RI7Ø		
IRET R_x		RI71		
JUMP ADDRESS		JMAS	P	
DJSR ADDRESS		JLAS	R_0	P
DJSR R_x, ADDRESS		JLAS	R_x	P
JPR R_x		JMAS	(R_x)	
JPR R_x + 1		JMAS	$+(R_x)$	
<u>Conditional Transfers</u>				
JIF R_x, NEG ADDRESS		JRFS	R_x	(R_{16})+Off
JAD R_x, ADDRESS		JR7S	R_x-	(R_{16})+Off
JAD R_y, R_x		JM7S	R_y-	(R_x)
IF R_x. GE. R_y SKIP		OP7B		
IF R_x. LT. R_y SKIP		OP7C		
IF R_x. EQ. R_y SKIP		OP7D		
IF R_x. NE. R_y SKIP		OP7E		
IF R_x, POS SKIP		JRFC	R_x	(R_{16})+SKIP*

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
IF R_x. EQ. 0 SKIP		JR2C	R_x	(R_{16})+SKIP
IF R_x. NE. 0 SKIP		JR2S	R_x	(R_{16})+SKIP
IF R_x(N). EQ. 1 SKIP		OP7F		
IF R_x(N). EQ. 0 SKIP		OPAØ		
IF ST(N), TRUE SKIP		OPA1		
IF ST(N), FALSE SKIP		OPA2		
<u>I/O Operations</u>				
IN R_x, DV - ADDR		MOVW	M_p	R_x
OUT R_x, DV - ADDR		MOVW	R_x	M_p
MASK R_x		MOVW	R_x	R_{18}
CALL DEBUGGER		IN7Ø		

*SKIP = Offset required to cause next command to be skipped.

10.6 OP CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Macro Coding		
$R_y = -R_x$	OP7Ø	MOVW	R_x	R_y
		COMW	R_y	1
$R_y = R_xF$	OP71	MOVW	R_x	R_y
		COMW	R_y	Ø
$R_y = R_x - R_y$	OP72	MOVW	R_x	R_{21}
		SBNW	R_{21}	R_y
		MOVW	R_{21}	R_y
R_y, R_x = R_y R_x	OP73	This is a fractional divide. R_y is divided by R_x, then multiplied by 2^{15}. The quotient is loaded into R_y.		
R_y, R_x = R_y R_x	OP74	This is double precision fractional divide. The numerator is the 32 bit quantity contained in R_y, Q. Rules of single precision divide are followed.		
R_y = -- R_x	OP75	MOVD	R_x	R_y
		COMD	R_y	1
$R_y = R_x + C$	OP76	MOVW	R_x	R_y
		ADNW	S	R_y
$R_y = R_x$ AND C	OP77	MOVW	R_x	R_y
		ANRW	S	R_y

ASSOCIATES, INC.

Mnemonic	MIS Mnemonic	Macro Coding		
$R_y = R_x$ OR C	OP78	MOVW ORRW	R_x S	R_y R_y
$R_y = R_x$ - C	OP79	MOVW SBNW	R_x R_y	R_y S
MTRAN Q WORDS FROM R_x TO R_y	OP7A	BLOCK, MOVW DECW JM2S JMAS	$(R_x)+$ RQ (R_{19})+SKIP BLOCK	$(R_y)+$
IF R_x. GE. R_y SKIP	OP7B	CMPW JM7S	R_x (R_{16})+SKIP	R_y
IF R_x. LT. R_y SKIP	OP7C	CMPW JM7C	R_x (R_{16})+SKIP	R_y
IF R_x. EQ. R_y SKIP	OP7D	CMPW JM2C	R_x (R_{16})+SKIP	R_y
IF R_x. NE. R_y SKIP	OP7E	CMPW JM2S	R_x (R_{16})+SKIP	R_y
IF R_x(N). EQ. 1 SKIP	OP7F	ANDW JM2C	MASK (R_{16})+SKIP	R_x
IF R_x(N). EQ. 0 SKIP	OPAØ	ANDW JM2S	MASK (R_{16})+SKIP	R_x
IF ST(N), TRUE SKIP	OPA1	Test bit N of the 256 bits of external status. If bit N is equal to Ø, execute the next instruction. If bit N is equal to 1, skip it.		
IF ST(N), FALSE SKIP	OPA2	Test bit N of the 256 bits monitoring external status. If bit N is equal to 1, execute next instruction. If bit N is equal to Ø, skip it.		

10.7 IN/RI CLASS INSTRUCTIONS

Mnemonic	MIS Mnemonic	Description
CALL DEBUGGER	IN7Ø	Generates a software interrupt.

SECTION 11

MICROPROCESSOR 8 - EADI

11.1 INTRODUCTION

Due to an extreme lack of documentation the major characteristics of the EADI microprocessor could only be surmised. One of its most apparent features is that it is closer in design to a custom bit-slice processor than a conventional microprocessor. It appears that the EADI utilizes 12 bit data words whereas addresses are specified by 16 bit words. The EADI appears to have four addressing modes and a direct-type of I/O control. Its interrupt structure cannot be determined from the information presently available.

11.2 PROGRAMMING MODEL

Registers	Designation (MIS)
A Register	R_0
B Register	R_1
C Register	R_2
Indirect Register	R_3
Program Counter	R_{16}
Processor Status Word	R_{17}
Carry	bit Ø
Equal	bit 2
Unconditional	bit A
MSB	bit C
Interrupt Register	R_{18}
Stack Pointer	R_{19}
Quotient Register	R_{30}

11.3 INTERRUPT STRUCTURE

The interrupt structure of the EADI can not be determined from the documentation presently available.

11.4 EXAMPLES OF ADDRESSING MODES

a. Direct

STAR RAM	MOVT	R_0, M

b. Indirect

STAI	MOVT	R_0, (R_3)

ASSOCIATES, INC.

c. Immediate

MASKA MASK, PGM, T	ANDT	#I, R_0
	JM2C	M
MASKA MASK, PGM, F	ANDT	#I, R_0
	JM2S	M

d. Conditional

FSKIP AEB	CMPT	R_0, R_1
	JM2S	(R_{16})+SKIP*
FJSUB PGM, AEB	CMPT	R_0, R_1
	JM2C	(R_{16})+SKIPO**
	MOVW	R_{16}, R_0
	MOVW	I/L***, R_{16}
	JMAS	M

11.5 TRANSLATION INTO MIS

Table 11.5.2 presents a list of the instruction set of the EADI microprocessor. The operands utilized by the EADI system are listed in Table 11.5.1. It should be noted that all instructions utilizing ALU operands are preceded by the macro-coding listed in Table 11.5.1. Thus the complete translation into the MIS consists of a macro code composed of the source instruction plus the appropriate preceding macro (****).

TABLE 11.5.1

KEY TO OPERAND(S) (OP)

ALU = ALU Code

DEC	MOVT	R_0	R_4
	DECT	R_4	
	S = R_4		
NA	MOVT	R_0	R_4
	COMT	R_4	Ø
	S = R_4		
NAND	MOVT	R_0	R_4
	ANRT	R_1	R_4
	COMT	R_4	Ø
	S = R_4		
MAX	S = #FFF		
NOR	MOVT	R_0	R_4
	ORRT	R_1	R_4
	COMT	R_4	Ø
	S = R_4		

NB	MOVT	R_1	R_4
	COMT	R_4	$\emptyset$
	S = R_4		
MINUS	MOVT	R_0	R_4
	SBNT	R_4	R_1
	S = R_4		
PLUS	MOVT	R_0	R_4
	ADNT	R_1	R_4
	S = R_4		
EXOR	MOVT	R_0	R_4
	EXRT	R_1	R_4
	S = R_4		
BR	S = R_1		
OR†	MOVT	R_0	R_4
	ORRT	R_1	R_4
LEFT	MOVT	R_o	R_4
	SL1T	R_4	
	S = R_4		
ZERO	S = #ØØØ		
ANDAB	MOVT	R_0	R_4
	ANRT	R_1	R_4
	S = R_4		
INCA	MOVT	R_0	R_4
	INCT	R_4	
	S = R_4		
AR	S = R_0		

DEV = Device Code

STAT	Status Bus
CIN	C Register
MUX4	Bit Ø to 3 of MUX Bus
MUX12	Bit 4 to 15 of MUX Bus
TROM	Trigonometric ROM

CC1 = Cond Code 1

U	Unconditional, b = A
Carry	Carry Flag, b = Ø
Equ	Equal Flag, b = 2
MSB	MSB of A Register, b = C

† S = R_4

CC2 = Cond Code 2

U	Unconditional, b = A
Carry	Carry Flag, b = Ø
Equ	Equal Flag, b = 2

RAM = RAM Address

PGM = Program Address

TABLE 11.5.2

TRANSLATION INTO MIS - EADI

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
<u>RAM Reference Instructions</u>				
STAR	RAM	MOVT	R_0	M
STAI		MOVT	R_0	(R_3)
STBR	RAM	MOVT	R_1	M
STBI		MOVT	R_1	(R_3)
SAVE	RAM	OP8Ø		
CLR	RAM	CLRT	M	
CLRI		CLRT	(R_3)	
PRE	RAM	MOVT	#FFF	M
PREI		MOVT	#FFF	(R_3)
ST	ALU	****		
		MOVT	S	M
STI	ALU	****		
		MOVT	S	(R_3)
LDAR	RAM	MOVT	M	R_0
LDAI		MOVT	(R_3)	R_0
LDBR	RAM	MOVT	M	R_1
LDBI		MOVT	(R_3)	R_1
<u>I/O Instructions</u>				
OUT	ALU, DEV	****		
		MOVT	S	M_p
OUTL	ALU, DEV	****		
		MOVT	S	M_p

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
OUTA	DEV	MOVT	R_0	M_p
OUTB	DEV	MOVT	R_1	M_p
IN	DEV	MOVT	M_p	R_1
<u>Branch Instructions</u>				
JSUB	PGM, U	OP81		
JSUB	PGM, CARRY	OP82		
JSUB	PGM, EQU	OP83		
JSUB	PGM, MSB	OP84		
STOP	U	IN8Ø		
STOP	CARRY	IN81		
STOP	EQU	IN82		
STOP	MSB	IN83		
JUMP	PGM, CC1	JMbs	P	
JUMP	PGM, CC1*****	JRCS	R_0	M
JOC	ALU, CC1	****		
		JMbs	(S)	
JOC	ALU, CC1*****	****		
		JRCS	R_0	(S)
JTP	CC1*****	JRCS	R_0	(R_0)
SKIP	CC1	JMbs	(R_{16})+Off	
SKIP	CC1*****	JRCS	R_0	(R_{16})+Off
RTN	RAM, CC2	JMbs	(M)	
FJUMP	PGM, AEB	OP85		
FJUMP	PGM, AEZ	OP86		
FJUMP	PGM, BEZ	OP87		
FJUMP	PGM, AEN	OP88		
FJUMP	PGM, BEN	OP89		
FJSUB	PGM, AEB	OP8A		
FJSUB	PGM, AEZ	OP8B		
FJSUB	PGM, BEZ	OP8C		
FJSUB	PGM, AEN	OP8D		
FJSUB	PGM, BEN	OP8E		
JTP	CC1	JMbs	(R_0)	

Mnemonic	Operand(s)	MIS Mnemonic	Operand 1	Operand 2
FSKIP	AEB	OP8F		
FSKIP	AEZ	OPBØ		
FSKIP	BEZ	OPB1		
FSKIP	AEN	OPB2		
FSKIP	BEN	OPB3		
MASKA	MASK	OPB4, 5		
MASKB	MASK	OPB6, 7		
<u>Arithmetic Logic Unit</u>				
DESTA	ALU	****		
		MOVT	S	R_0
DESTB	ALU	****		
		MOVT	S	R_1
DESTC	ALU	****		
		MOVT	S	R_2
SETA	Const	MOVT	#I	R_0
SETB	Const	MOVT	#I	R_1
SRT		OPB8		
SLFT		OPB9		
SFLAG	ALU	****		
		SLØT	SD	

* SKIP = Offset required to cause the next instruction to be skipped.

** SKIPO = Offset required to cause a jump out of the macro code.

*** I/L = Because of insufficient documentation it was not possible to determine exactly how a jump to subroutine is accomplished. According to the available documentation the content of the program counter (R_{16}) is saved in R_0. I/L is then transferred to R_{16}. Unfortunately the meaning of I/L is not known, so it could not be translated into an MIS operand.

**** = EADI commands utilizing ALU operands must be preceded by the appropriate macro listed under ALU = ALU Code in Table 11.5.1.

***** = The case where CC1 = MSB.

++ =
```
MOVW    R16   R0
MOVW    I/L   R16
JMAS    M
```

ASSOCIATES, INC.

11.6 OP CLASS INSTRUCTIONS

Mnemonic	Operand(s)	MIS Mnemonic	Macro Coding		
SAVE	RAM	OP8Ø	MOVT	R_0	M
			INCT	M	
JSUB	PGM, U	OP81	MOVW	R_{16}	R_0
			MOVW	I/L***	R_{16}
			JMAS	M	
JSUB	PGM, CARRY	OP82	JMOC	(R_{16})+SKIPO	
			++		
JSUB	PGM, EQU	OP83	JM2C	(R_{16})+SKIPO	
			++		
JSUB	PGM, MSB	OP84	JRCC	R_0	(R_{16})+SKIPO
			++		
FJUMP	PGM, AEB	OP85	CMPT	R_0	R_1
			JM2S	M	
FJUMP	PGM, AEZ	OP86	JR2S	R_0	M
FJUMP	PGM, BEZ	OP87	JR2S	R_1	M
FJUMP	PGM, AEN	OP88	CMPT	#FFF	R_0
			JM2S	M	
FJUMP	PGM, BEN	OP89	CMPT	#FFF	R_1
			JM2S	M	
FJSUB	PGM, AEB	OP8A	CMPT	R_0	R_1
			JM2C	(R_{16})+SKIPO	
			MOVW	R_{16}	R_0
			MOVW	I/L	R_{16}
			JMAS	M	
FJSUB	PGM, AEZ	OP8B	JR2C	R_0	(R_{16})+SKIPO
			MOVW	R_{16}	R_0
			MOVW	I/L	R_{16}
			JMAS	M	
FJSUB	PGM, BEZ	OP8C	JR2C	R_1	(R_{16})+SKIPO
			MOVW	R_{16}	R_0
			MOVW	I/L	R_{16}
			JMAS	M	

Mnemonic	Operand(s)	MIS Mnemonic	Macro Coding		
FJSUB	PGM, AEN	OP8D	CMPT	#FFF	R_0
			JM2S	(R_{16})+SKIP0	
			MOVW	R_{16}	R_0
			MOVW	I/L	R_{16}
			JMAS	M	
FJSUB	PGM, BEN	OP8E	CMPT	#FFF	R_1
			JM2S	(R_{16})+SKIP0	
			MOVW	R_{16}	R_0
			MOVW	I/L	R_{16}
			JMAS	M	
FSKIP	AEB	OP8F	CMPT	R_0	R_1
			JM2S	(R_{16})+SKIP	
FSKIP	AEZ	OPBØ	JR2S	R_0	(R_{16})+SKIP
FSKIP	BEZ	OPB1	JR2S	R_1	(R_{16})+SKIP
FSKIP	AEN	OPB2	CMPT	#FFF	R_0
			JM2S	(R_{16})+SKIP	
FSKIP	BEN	OPB3	CMPT	#FFF	R_1
			JM2S	(R_{16})+SKIP	
MASKA	MASK, PGM, T	OPB4	ANDT	#I	R_0
			JM2C	M	
MASKA	MASK, PGM, F	OPB5	ANDT	#I	R_0
			JM2S	M	
MASKB	MASK, PGM, T	OPB6	ANDT	#I	R_1
			JM2C	M	
MASKB	MASK, PGM, F	OPB7	ANDT	#I	R_1
			JM2S	M	
SRT		OPB8	SR1T	R_0	L
			SR1T	R_2	L
SLFT		OPB9	SL1T	R_0	
			SL1T	R_2	

11.7 IN/RI CLASS INSTRUCTIONS

Mnemonic	Operand(s)	MIS Mnemonic	Description
STOP	U	IN85	Unconditional stop
STOP	CARRY	IN86	Conditional stop
STOP	EQU	IN87	Conditional stop
STOP	MSB	IN88	Conditional stop

APPENDIX D:

REPORT OF A TASK STUDY ON SOFTWARE DEVELOPMENT AIDES FOR THE U.S. ARMY ADVANCED ATTACK HELICOPTER FIRE CONTROL SYSTEM

This page was left blank intentionally.

ASSOCIATES, INC.

APPENDIX D

REPORT OF A TASK STUDY ON SOFTWARE DEVELOPMENT AIDES IN U. S. ARMY ADVANCED ATTACK HELICOPTER (AAH) FIRE CONTROL SYSTEMS

1. INTRODUCTION

This study of software development aides was conducted in the context of the Advanced Attack Helicopter (AAH) Fire Control System and as required by the scope of work of the subject contract. Presently the fire control system is designed using fourteen embedded microprocessors of nine different types. The large number of computer types and computing languages make future maintenance very difficult and expensive. The U.S. Army is considering replacement of these embedded microprocessors by a single microprocessor, henceforth referred to as the common-microprocessor. The evolution from the original microprocessors to the common-microprocessor is envisaged as proceeding in three phases. In the first phase the replacement standard-microprocessor will be equipped with microprograms of the respective original microprocessors and will be capable of emulating the programs in the original microprocessors. In the second phase the assembly language programs of the original microprocessors will be translated automatically into the assembly language of the standard-microprocessor. This will enable maintenance of the programs, originally created in multiplicity of assembly languages, using uniform assembly language of the standard microprocessor. The maintenance of these programs will thereby require far less expertise and would be less costly. Finally, the third phase will be oriented to future development of new or replacement programs. The software development would be further simplified and less costly by availability of a High Level Language compiler for the common-microprocessor, in which the new programs will be composed. This study is concerned with software development aides such as the High Level Language compiler as well as an assembler, debugger, simulater and link editor, which will all be valuable in development of future programs.

ASSOCIATES, INC.

These software development aides will have to be endowed with certain capabilities which are above and beyond those required in general for computer program development. These special capabilities originate from the real time and efficiency requirements of the fire control system. In particular, the compiler for the High Level Language would be required to produce highly optimized code which provides faster execution time and reduced memory requirements. Also, a simulator would be very helpful to allow development and debugging of programs on a larger computer system, thereby speeding the program development process. The assembler and link editor are assumed not to pose any special problems and are not discussed here further.

The remainder of this appendix is devoted to discussions of two areas:

1) Selection of a High Level Language

2) Optimizer characteristics of the computer

The technique for selection of a High Level Language and for construction of an optimizing compiler represent known state of the art. Therefore, the respective development does not appear to represent significant risks. Generally, five man-years of effort over one and one half years would be required.

2. SELECTION OF A HIGH LEVEL LANGUAGE

Probably the foremost requirement of a High Level Language is that it be widely used. Thereby there would be a large community of individuals proficient in the language who would not require specialized training to be able to maintain or compose programs. The language therefore should also be recognized as a standard by the American National Standards Institute or by the Department of Defense. A second requirement concerns the availability of facilities in the language that will make programming easier and less costly. However, due to the rapid changes in computer languages it is rather difficult to establish a clear advantage of one language over another in this latter respect. For example, FORTRAN is the oldest established High Level Language, however, the new 1977 standard of FORTRAN includes many facilities which make it competitive with the more recently introduced languages. PASCAL is a newly introduced language which recently has found particular favor in use for microprocessors. PASCAL is currently going through the process of standardization and there are several versions for this language with increasing capabilities. ADA is a recently proposed programming language intended as a standard for the Department of Defense, for which there has not been a compiler so far. This dynamic situation with the most important High Level Language would make it difficult to select a High Level Language. However, any of the above would be adequate.

There are a number of other considerations which may be necessary to take into account in evaluating and selection of High Level Languages. Some of these are listed below.

1) Correcting programs by default: The underlying philosophy in some languages is that if a user omits certain information in statements the compiler then introduces default values to complete the program statements. In other languages, this is being objected to on the ground that the formalism of the semantics of the program is thereby undermined. An example, of these opposing approaches are FORTRAN, where data types are assumed if not specified by the user, and PASCAL where data types must be specified or otherwise an error message is issued.

2) Memmory allocation: In some languages, the amount of memory required for data is determined in the program statements themselves. An example of this type of language is FORTRAN where the amount of memory allocated to data is fixed at the time of the compilation of the program. In other languages, the amount of memory needed may be determined at run time and acquisition or release of memory space can be performed dynamically at run time. Certain versions of PASCAL and in the future ADA are examples of the latter philosophy.

3) Size of compiler: Generally the more powerful the language the larger would be the compiler. If it is required to run the compiler on the same microprocessor that is used for the application then it will be very important that the compiler would be simple and have limited memory space requirements. For instance, FORTRAN 77 is a very large compiler and may therefore be too large. By comparison PASCAL compilers have been much smaller, which has made PASCAL such a favorite for programming microprocessors.

4) Dependence on operating system features: Some programming languages depend on a certain commands that are performed by an operating system and also require incorporating in the program routines which support the various statements. This would necessarily increase the memory requirements of the programs generated by such a compiler. Again, PASCAL is an example where the programs generated could be independent of functions of operating systems or of libraries of routines that must be incorporated in the program.

5) Special programming features: There is a variety of other features which are important in facilitating the programming process in some applications. Several such features are listed below.

Data Structure declarations are available both in FORTRAN 77 and in PASCAL. As noted, the data type specifications are available in PASCAL but not in FORTRAN. Additionally, PASCAL allows user specifications of data types. Recursive capability is available in PASCAL but not in FORTRAN. Some languages have very powerful capability for manipulating character strings and for structured programming constructs. Both FORTRAN 77 and PASCAL possess these features. Finally it is very important to be able to accommodate the variety of different file organizations of input-output data. Again both FORTRAN 77 and PASCAL include such facilities.

3. COMPILER OPTIMIZATION

In addition to the normal compilation, it will be necessary to include in the compiler optimization processes that will reduce program memory and execution time requirements. Optimization techniques can basically be divided into two types. First, where the source program statements provided by the user are modified with the objective of optimizing the program. Second, optimal instruction sequences are selected to represent the various constructs in the High Level Language to assure efficient utilization of instructions and registers in the object machine.

Modifying the source code: The techniques used in this process include the following:

1) The order of the statement in the program may be modified to make the program more efficient: For example, a computation statement in an iteration loop which is independent on the iteration parameter can be moved out of the loop and placed ahead of it. Also scope of certain iteration loops may be optimized by merging wherever possible loops with the same iteration parameters.

2) Modifying individual statements to reduce necessary computation: For instance, this involves recognizing common sub-expressions in a statement and modifying the statement so that the sub-expression is computed only once. Commutativity of operators can be utilized to reduce the number of operations. Operations which can be done during compile time could be eliminated altogether. Sometimes, it may be effective to replace a multiplication with additions, and division with subtractions.

3) Consolidation and deletion of statement: This would include removing any code which is not referred. If a procedure or a function is called only once then the call for the procedure or function may be eliminated.

4) Minimizing the data storage requirements: This includes determining which variables do not require space simultaneously and therefore can share memory space.

The gereration of an efficient machine code to represent High Level Language includes some global and local optimization techniques. The approach used is first to generate high level code which takes advantage of the target machine architecture and then to generate machine code for the object machine. Storage savings can be achieved by allocating working registers wherever possible. Special computer architecture further may be utilized. For instance, stuck operations are important in computing expressions. This class of optimization techniques is very similar to those described in Appendix B where the process of translation of assembly language code was described. Therefore, the optimization techniques are not described here in further detail.

DISTRIBUTION LIST

Commander	
US Army Armament Research and Development Command.	
Attn. DRDAR-SCF-DA	12
DRDAR-QAF-R	2
DRDAR-QAF-A	2
DRDAR-TSS	5
DRDAR-SC	1
DRDAR-SCP	1
DRDAR-SF	1
Dover, N. J. 07801	
Defense Technical Information Center	12
Cameron Station	
Alexandria, VA 22314	
Weapon Systems Concept Team/CSL	1
Attn. DRDAR-ACW	
Aberdeen Proving Ground, MD 21010	
Technical Library	1
Attn. DRDAR-CLJ-L	
Aberdeen Proving Grounds, MD 21010	
Director	1
US Army Ballistics Research Laboratory	
ARRADCOM - Attn DRDAR-TSB-S	
Aberdeen Proving Grounds, MD 21010	
Benet Weapons Laboratory	1
Technical Library	
Attn. DRDAR-LCB-TL	
Watervliet, NY 12189	
Commander	1
US Army Armament Material Readiness Command	
Attn. DRSAR-LEP-L	
Rock Island, IL 61299	
US Army Material Systems Analysis Activity	1
Attn. DRXSY-MP	
Aberdeen Proving Grounds, MD 21005	