SAMSO TR 71-6

ARCHITECTURAL STUDY FOR ADVANCED GUIDANCE COMPUTERS

Part 2

GUIDANCE COMPUTER ARCHITECTURE STUDY

Final Report

Contract F04701-70-C-0065

5 February 1971

Prepared for
Space and Missile Systems Organization (SAMSO)
United States Air Force Systems Command

by

CIRAD
401 NORTH HARVARD
CLAREMONT, CALIFORNIA 91711
(714) 621-3942

CDRL A007

WS-1007-3-6

This document has been approved for public release and sale; its distribution is unlimited

314

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

CIRAD
401 North Harvard
Claremont, CA. 91711

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

3. REPORT TITLE

Architectural Study for Advanced Guidance Computers

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Final Report (February 1970 through December 1970)

5. AUTHOR(S) *(Last name, first name, initial)*

Wersan, Stephen J.; Colen, Paul; Carey, Levi; Trout, Robert

6. REPORT DATE

6 February 1971

7a. TOTAL NO. OF PAGES

7b. NO. OF REFS

5

8a. CONTRACT OR GRANT NO.

F04701-70-C-0065

b. PROJECT NO.

Program 672A

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)

WS-1007-3-6

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

SAMSO TR 71-6

10. AVAILABILITY/LIMITATION NOTICES

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

HQ SAMSO (SYGN)
Norton AFB, CA. 92409

13. ABSTRACT

The objective of the study was to define an advanced guidance computer architecture that will permit the effective use of high-order programming languages in the definition and implementation of advanced ballistic missile missions.

Part 1 of the Final Report entitled "Guidance Programming Language Study", presents the specification of a high-order programming language suitable for programming advanced guidance and targeting missions for the advanced guidance computer architectures. A subset of the Space Programming Language (SPL/J6) was selected and improved, and its syntactic forms analyzed for efficient code generation for the architectures under consideration.

Part 2 of the Final Report entitled "Guidance Computer Architecture Study", contains the selected architecture together with the SPL language and compiler considerations involved in the design, and the programming tradeoff studies. The study placed emphasis on the ability of the architecture to efficiently execute compiler generated code.

A selected set of guidance and targeting equations was used as a vehicle for conducting tradeoff studies. SPL compiler generated code forms were studied for interfacing with computer functions. The size efficiency of the object code compared to that of assembly programming for traditional single address fixed point airborne computer architectures was the major design consideration. The resulting architecture is effective in satisfying other functional guidance computer system requirements (i.e. execution time and memory size) while significantly improving the size effiency of generated object code.

Security Classificatior

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer architecture						
Computer design						
Hardware/software compatibility						
Higher order languages						
Space Programming Language (SPL)						
Compiler generated code						
Guidance computers						
Stack machine organization						
Polish notation						
Array processors						
Syntactic language anaysis						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through

________________________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through

________________________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through

________________________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS)*, *(S)*, *(C)*, or *(U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

UNCLASSIFIED
Security Classification

SAMSO TR 71-6

ARCHITECTURAL STUDY FOR ADVANCED GUIDANCE COMPUTERS

Part 2

GUIDANCE COMPUTER ARCHITECTURE STUDY

Final Report

Contract F04701-70-C-0065

5 February 1971

Prepared For
Space and Missile Systems Organization (SAMSO)
United States Air Force Systems Command

by

Corporation for Information Systems
Research and Development
(CIRAD)
401 N. Harvard
Claremont, California 91711
(714) 621-3942

CDRL A007

WS-1007-3-6

This document has been approved for public release and sale; its distribution is unlimited

ABSTRACT

This Volume, Part 2 of the Final Report for the Architectural Study for Advanced Guidance Computers, represents the results of a study to specify the architecture of missileborne computers which will permit the effective use of higher order programming languages to define and implement advanced guidance mission programs. The Space Programming Language (SPL) was selected and improved, and its syntactic forms analyzed for efficient compiler implementation. Utilizing a selected set of guidance and targeting equations as a vehicle for conducting tradeoff studies, compiler code forms were studied for interfacing with computer functions. The resulting architecture is effective in satisfying other computer system requirements while significantly improving the size efficiency of generated object code.

FORWARD

This is the final report on the Architectural Study for Advanced Guidance Computers. This effort was performed for SAMSO, Air Force Systems Command, USAF, under contract number F04701-70-C-0065, over a ten month period from 15 February to 15 December 1970.

The objective of the study is to define an advanced guidance computer architecture that will permit the effective use of high-order programming languages in the definition and implementation of advanced ballistic missile missions.

This Volume, Part 2 of the Final Report, entitled Guidance Computer Architecture Study, contains the selected architecture together with the SPL language and compiler considerations involved in the design, and the programming tradeoff studies. The study placed emphasis on the ability of the architecture to efficiently execute compiler generated code. The size efficiency of the object code compared to that of assembly programming for traditional single address fixed point airborne computer architectures was the major design consideration.

The other volume of the final report, Part 1, entitled Guidance Programming Language Study, contains a syntactic analysis of the Space Programming Language (SPL/MK II), the revisions and extensions to create SPL/MK III, and the verified metalinguistic definition of SPL/MK III in TBNF.

CIRAD wishes to acknowledge the excellent technical support provided by its consultants, and the guidance and assistance provided by both SAMSO and The Aerospace Corporation throughout the course of this study. Part 1 of the final report was prepared through a joint effort between CIRAD and CSA, Inc. The authors for Part 1 are Dr. Stephen J. Wersan and Mr. Paul Colen from CIRAD, and Mr. Levi Carey from CSA, Inc. Consulting assistance was provided by Mr. Robert Trout. The authors of Part 2 of the final report are Dr. Stephen J. Wersan, and Mr. Paul Colen from CIRAD, with consulting assistance from Mr. Robert Trout, and Mr. Levi Carey.

This Technical Report has been reviewed and is approved.

Paul Colen

Paul Colen,
Program Manager,
CIRAD

This Technical Report has been reviewed and is approved.

Ronald J. Starbuck, Captain USAF
SAMSO/SYGN Project Officer

TABLE OF CONTENTS

Page

1.0 Introduction 1-1

2.0 Summary of Results 2-1

3.0 Advanced Guidance Computer (AGC) Architecture Summary 3-1

3.1 Design Rationale and General Systems Description 3-1
3.2 Arithmetic and Control Function Summary 3-35
3.3 Data Representation 3-43
3.4 Memory Allocation 3-44
3.5 Program String Syllables 3-45
3.6 Descriptor Summary 3-48
3.7 File Descriptor 3-50

4.0 Architecture Description 4-1

4.1 Memory Structure and Addressing 4-1
4.2 Numeric Data and Arithmetic 4-7
4.3 Instruction Fetch Cycle 4-14
4.4 The Stack Mechanism 4-21
4.5 Instruction Syllables - Scalar Mode 4-27
4.6 Partial Word Operations 4-46
4.7 Indexing 4-58
4.8 Program Segmentation and Transfer of Control 4-69
4.9 Procedures Entry and Return 4-78
4.10 Array Mode Processing 4-91
4.11 Interrupt System 4-112
4.12 Circumvention 4-124
4.13 Input/Output 4-128

5.0 Programming Studies 5-1

5.1 Programming Summary 5-2
5.2 Programming Examples 5-6

6.0 Additional Considerations 6-1

6.1 Existing SPL/MK III Intrinsic Functions 6-1
6.2 Proposed Additional SPL/MK III Intrinsic Functions 6-2
6.3 Proposed Additional Operators 6-6

7.0 References 7-1

LIST OF FIGURES

Figure		Page
4-1	AGC Memory Map	4-6
4-2	Instruction Fetch Registers	4-15
4-3	Instruction Fetch Cycle	4-18
4-4	Example of Syllables in Memory	4-19
4-5	Instruction Fetch Registers During Execution of Syllables Shown in Figure 4-4	4-20
4-6	Operand Fetch	4-23
4-7	Binary Composition	4-25
4-8	EXT and SXT Shifting Steps	4-54
4-9	INS and SNS Shifting Steps	4-56
4-10	Execution of Transfer Operators	4-77
4-11	Illustrated Stack Unwinding in Procedure Returns	4-87
4-12	Format of Array Descriptor	4-98
4-13	Flowchart of Cross Product Fetch (XPF)	4-100
4-14a	Before Array Step 2	4-105
4-14b	After Array Step 2	4-106
4-14c	After Array Step 3	4-107
4-14d	After Array Step 4	4-108
4-15	Detail Trace of Array Step 4	4-110
4-16	Initiation of Interrupt	4-115
4-17	File Descriptors	4-132

LIST OF TABLES

Table		Page
2-1	AGC Architecture Summary of Programming Efficiency	2-1
3-1	Mneumonic Definitions	3-21
4-1	Control Stream Syllables Types	4-28
4-2	Scalar Operations, Arithmetic Functions	4-31
4-3	Scalar Operations, Boolean, Logical Functions	4-32
4-4	Scalar Operations, 'Comparison and Branching Function	4-33
4-5	Scalar Operations, Interrupt Processing	4-34
4-6	Scalar Operations, Storing and Indexing	4-35
4-7	Scalar Addressing Operators	4-40
4-8	Scalar Mode Descriptor Calls	4-43
4-9	Stack Representations of Subroutine Arguments	4-80
4-10	Columns vs Row Fetch in Array Operations	4-94
4-11	Array Arithmetic Operators	4-96
4-12	Miscellaneous Array Operations	4-97
4-13	Array Mode Control Registers and Indicators	4-99
5-1	Programming Studies - Memory Requirements Summary	5-5

1.0 Introduction

The primary objective of the Architecture Study for Advanced Guidance Computers was to develop an advanced guidance computer architecture together with a compatible guidance system programming language which would permit the effective use of the programming language for definition and implementation of advanced guidance missions. The study set out to show how with effective interplay between compiler writers and machine designers, an architecture could be evolved which would permit compiler generation of efficient object code for the selected architectural configuration.

The first step in the effort was to select a suitable high order programming language for guidance system programming. A version of the Space Programming Language (SPL/MK II) was selected. A careful syntactic analysis of the language was conducted, and obvious ambiguities and inconsistencies which would impede efficient object code generation for any machine architecture were resolved. Next extensions and revisions to the language were proposed to make the language more effective for advanced guidance programming. Each language form was carefully studied from both the user's viewpoint, and in relation to the type of code that could be produced by an effective compiling process. The subset of SPL selected together with the revisions and extensions comprised a definition of the language (SPL/MK III), the syntax of which was verified by developing a metalinguistic definition (TBNF), and finally generating architecture is necessary to verify the initial design, to conduct timing tradeoff studies, and to verify the sizing estimates derived under the current effort.

The design rationale for the architecture is presented in Section 3, together with a summary of the salient features of the selected architecture. The summary includes a description of the arithmetic/control functions, the data representation, memory allocation, the program string syllable structure, and the control descriptors.

An extensive description of the architecture is presented in Section 4. In addition to the architectural description, the language and compiler implications are presented together with numerous source program and object code examples.

The programming tradeoff studies are presented in Section 5. An analysis is presented for each of the guidance and targeting equations and functions, and a summary created of the program string required to execute the equations on the selected architecture. The SPL/MK III source program is next presented. The advanced guidance computer architecture (AGC) code which resulted from hand compilation of the SPL/MK III source program is presented. This is followed by the assembly language coding for the IDCU.

Section 6 presents some additional considerations in relation to the guidance programming language. These considerations were developed late in the study effort and resulted from code optimization considerations in the final stages of the programming studies.

2.0 Summary of Results

The study has clearly demonstrated the feasibility of providing an Advanced Guidance Computer Architecture which solves the problem which currently plagues the aerospace compiler user, that of grave inefficiencies introduced into the operational program through code expansion from compiler generated code. By considering both the characteristics of the high-order language, in this instance the Space Programming Language (SPL/MK III), and the aspects of the compiling process during the functional design process for the airborne computer, compatibility has been achieved between the high-order language and the selected architecture.

The results of the programming studies clearly indicate the superiority of the selected architecture over both a traditional single address airborne computer architecture and an improved airborne processor such as the IDCU. A set of 24 equations was used throughout the study. This certainly represents a sufficient sample size to clearly indicate an efficiency trend.

A summary of the programming efficiency for the AGC architecture compared to the ATS single address baseline and to the IDCU is presented in Table 2-1. There is an overall reduction of 60% in the memory requirements for the AGC architecture vs a traditional single address architecture for implementing the same set of guidance equations and functions. This percentage holds for both the in-line code and for the service routines as a whole. This relationship

There is no doubt that the significant increase in compiler programming efficiency for the AGC architecture is achieved at the expense of additional processor control logic. The availability of large scale integration (LSI) microelectronics is rapidly reducing the size, weight, power, and reliability, penalties for increases in control complexity. There are at present 42 arithmetic and control functions identified for the AGC, 26 of which appear to be probable candidates for implementation in registers and flip-flops rather than in dedicated memory locations. The selection of these functions for register implementation was in most instances done on a feeling of what functions would need to be implemented as high speed operations to sustain computation efficiency. A more effective means of setting computational performance and control functional implementation location is through simulation of the proposed architecture.

There are several indications that the selected architecture will execute the selected equations and functions well within the specified execution timing requirements. The architecture has a form of pipe-lining for the program string. This approach minimizes instruction access time. The operand/operator string reduces memory accesses thereby increasing effective execution time. The architecture incorporates an automated way of handling the stack operations between the arithmetic registers and the stack in scratch pad memory. Program constants are carried as literals in the program string and are transferred directly to the stack further reducing memory access time. Finally, some programming language functions, for example loop-control, are directly implemented in the architecture.

holds for the in-line code for each of three different subsets (figures) of the representative equations and functions.

There is an overall reduction of 40% in the memory requirements for the AGC architecture vs the IDCU for programming the same set of guidance and targeting functions. For one subset the reduction was 30% whereas for the other subset the reduction was 55%. The AGC's efficiency is due to the use of a polish stack with implied addressing, the use of floating point, the number representation used, direct fetch of literals from instructions, built-in array operations and use of 1 or 2 byte instructions without word-boundary restrictions.

It is recognized that further analysis might reduce the differences in programming efficiency between the three architectures. On the one hand one could argue that the 20% allowance for such functions as shifting, scaling and growth for the sizing done for the ATS configuration could be reduced by careful reprogramming of the equations and functions. On the other hand the AGC program string includes many constants which in the single address configuration are stored and addressed as data. The size of the equation and function sample has resulted in a number of data points which make the probability of reversing the indicated efficiency trend very small.

ADVANCED GUIDANCE COMPUTER ARCHITECTURE
SUMMARY OF PROGRAMMING EFFICIENCY

	AGC Memory Words	ATS Memory Words	AGC Memory Words	IDCU Memory Words
	Figure 2,3,4	Figure 2,3,4	Figure 1,5	Figure 1,5
In-Line Program	76	177	55	106
Service Routines	80	193	32	32
Totals	156	370	87	138

All word lengths are 32 bits. Figure numbers refer to Section 5.

There is no doubt that the significant increase in compiler programming efficiency for the AGC architecture is achieved at the expense of additio processor control logic. The availability of large scale integration (LSI) microelectronics is rapidly reducing the size, weight, power, and reliability, penalties for increases in control complexity. There are at present 42 arithmetic and control functions identified for the AGC, 26 of which appear to be probable candidates for implementation in registers and flip-flops rather than in dedicated memory locations. The selection of these functions for register implementation was in most instances done on a feeling of what functions would need to be implemented as high speed operations to sustain computation efficiency. A more effective means of setting computational performance and control functional implementation location is through simulation of the proposed architecture.

There are several indications that the selected architecture will execute the selected equations and functions well within the specified execution timing requirements. The architecture has a form of pipe-lining for the program string. This approach minimizes instruction access time. The operand/operator string reduces memory accesses thereby increasing effective execution time. The architecture incorporates an automated way of handling the stack operations between the arithmetic registers and the stack in scratch pad memory. Program constants are carried as literals in the program string and are transferred directly to the stack further reducing memory access time. Finally, some programming language functions, for example loop-control, are directly implemented in the architecture.

It is easy to conclude that by utilizing a 2 microsecond cycle time in NDRO memory coupled with T^2L MSI/LSI circuitry the selected architecture will satisfy the necessary computational requirements. However, it is very difficult to determine, without simulating the architecture, which control features must be implemented in circuitry, and which functions could best be located in memory in order to minimize control circuitry, while maintaining the computational efficiency necessary to meet the timing requirements.

The second objective of the study was achieved with the creation of a verified syntax for the Space Programming Language (SPL/MK III). All known ambiguities and inconsistencies were removed from the selected subset of SPL, necessary revisions to selected language elements accomplished, desirable extensions to language elements were proposed, and a metalinguistic definition of the recommended guidance programming language (SPL/MK III) was created and verified.

Finally, the compiling process either accomplishes the memory allocation process, or accomplishes partial memory allocation and furnishes information to the object time loader to complete memory allocation in a dynamic manner at object time.

A commonly used form of the intermediate language for the arithmetic and logical portions of the source input is polish notation. The original notation has been successfully expanded to include many other functions, for example, procedures and subroutines calls by name and value. The architecture selected executes a left-hand polish operand/operator string which is a direct transformation of the intermediate language. There are several features of the operand/operator string worth noting at this time. The polish string concept minimizes storage requirements for intermediate results for arithmetic and logical equations. The natural embodiment of this concept in machine architecture is through a last-in-first-out (LIFO) stack. In this manner, the scalar operators utilize either the top two items in the stack (binary) or the top item in the stack (unary) leaving the result in the top of the stack. This is frequently termed implicit addressing, and results in considerable savings in addressing operations.

Operands such as constants and program addresses, which are known as the program string is generated, are directly included in the program string as literals. This is quite natural to the compiling process. The literals are transferred directly from the program string to the top of the stack. This is

3.0 Architecture Summary

3.1 Design Rationale and General System Description

3.1.1 Design Rationale

The architecture under consideration has been evolved from an analysis of the language elements that comprise SPL/MK III together with the compiling processes necessary to transform SPL source program into object code. The SPL language and compiler considerations have been included as part of the architecture description throughout Section 4. In most instances the language considerations, and the compiling process, are discussed and then illustrated with examples.

In a simplified sense the compiling process may be functionally described as scanning the source program and translating the source input string into an intermediate language. The intermediate language is highly compact in relation to the source string. The information concerning data items, variables, constants, arrays, is placed in Tables. The information concerning program control, and information concerning allocation, is placed in other Tables. Utilizing the information contained in the various Tables, the compiling process then shifts to generating object code.

First the program string is generated for each of the parsed source statements. Modern compilers generate invariant code which is relocatable without modification by the allocation process. The compiler process next utilizes the control information, for example, procedure references, subroutine and function calls, and loop control, to interconnect the program strings.

another form of implicit addressing and again results in savings in addressing operations. The remainder of the operands are obtained by placing relative addressing references to their memory locations in the program stream.

The selected architecture utilizes the table created by the compiling process for program control, and the table of descriptions of data arrays as part of its control structure. These tables are stored in the form of descriptors in a portion of the program memory called the program control table (PCT). The architecture utilizes the program control table to control the flow of the object program. This implementation considerably reduces the amount of compiler generated code to accomplish these functions. In addition it simplifies and reduces the inefficiencies associated with the memory allocation process.

The nature of the airborne guidance problem suggests that memory allocation be completed in the compilation process rather than by the object time loader. The rationale for this is based upon the necessity for the airborne program to reside in a memory which cannot be altered in the operational environment, and the lack of auxilary storage necessary for dynamic storage allocation in the operating environment. The compiling process would, therefore, assign the base addresses for each program segment and the base address for each data array and input/output area in the appropriate descriptors in the program control table to complete the allocation process. For a typical guidance program the completion of the allocation process would involve assigning less than a thousand actual addresses.

In developing the operand/operator scheme for the architecture, a great deal of attention was given to the information content of the program string. It was determined that all scalar operators could be accommodated within a byte (8-bits), and that all direct addressing operations could be accomplished within 2 bytes. Literals in the program string could be accommodated in most cases with 1 byte, and in virtually all cases, within 2 bytes. Transfers of control between program segments, initiating of input/output operations, and the first reference to each data array could be handled by a 2-byte indirect addressing operation. This resulted in an extremely compact program stream consisting of 2-byte addressing syllables, one-byte operator syllables, and one or two byte literal syllables. The mechanization of the program string utilizes the first 2-bits in a syllable to determine the syllable type. In order to eliminate unusable portions of 32-bit words, which would otherwise greatly reduce the efficiency of the program syllable stream, a design has been provided such that addressing syllables and 2-byte literal syllables may cross 32-bit word memory boundaries.

Considerable attention was given to the information content of the addressing operations. The concept of implicit addressing through the use of scalar and array stacks has already been mentioned. The explicit addressing structure gives consideration to the radiation environment, which requires that scalar variables, arrays, and stacks be separated from the program segments. The concept of partial addressing was utilized throughout the design. Provisions were made to divide the scratch pad and program storage into segments. The size of each segment is set by the allocation process of the compiler. The base location of each segment type is assigned by the allocation process to base address registers which can be either dedicated locations in scratch pad memory or addresses contained in program control storage. For example, scalar variables are accessed relative to the base address of scratch pad memory (SPM), the stacks are accessed via the stack pointer registers (P_0, P_1, and P_2), each program control table through the program control table address (PCT), and each program segment through the program segment base register (PSB).

The direct addressing syllable can directly address scalar variables and scratch pad memory and the program control table. It can also directly address any control stream syllable or local constant in a program segment. This addressing scheme coupled with the implicit addressing associated with stack operation covers the vast majority of the addressing operations in program execution.

The indirect addressing syllable accesses descriptors in the program control table which in turn contain the base address of program segments, the base address of input/output areas, and the base address of data arrays. Information from the addressing syllable is combined with information contained in control descriptors to perform at object time complex addressing operations which would otherwise be inefficiently performed by compiler generated code.

The selection of the architecture considered the efficient implementation of SPL procedure execution and of subroutine and function entry and exit. Careful attention has been given to the efficient passing of parameters to subroutines and functions and to the return of results upon subroutine or function completion.

The utilization of stacks is an integral part of the proposed architecture. The architecture contains three stacks, one for scalar operations, and two additional stacks for array operations. The scalar stack is a combination of a portion of scratch pad memory coupled to the arithmetic registers, whereas the additional array stacks are lists in scratch pad memory. A method has been provided to automatically handle adjusting the top of the stack and to minimize memory references for scalar stack operation.

A uniform method of arithmetic representation has been selected for the proposed architecture. The representation is capable of accomplishing the potential advanced guidance and targeting equation and functions for advanced guidance missions within the accuracy requirements, without the use of fixed point operations. The arithmetic representation selected is a floating point representation whereby a zero exponent represents arithmetic integers. This representation was selected to eliminate the inefficiencies involved in generating code for fixed point and mixed arithmetic operations. Additional floating point control functions have been provided to terminate computations on boundary conditions.

The architecture provides for partial word operations for handling the formatting and conversion of input and output quantities, input and output character or bit streams, and output telemetry data. The partial word insertion and extraction operations are based on the concept of identifying the starting bit and field length of the partial word data.

Indexing operations in the AC architecture have been separated into two functional areas, indexing for program loop control, and indexing for handling data arrays. This separation permits efficient handling of the indexing function by the compiler. Indexing operations on arrays are handled in the array mode by the array operators, whereas accessing a data element from an array is handled by an addressing operation which involves an index value in the stack and the array descriptor. Indexing operations for loop control are handled by a generalized indexing function which combines an increment with a current value to test against a final value upon the completion of each iteration of the loop. For nested loops the higher level loop control values are stored in the scalar stack.

An array processing mode has been provided in the selected architecture. The mode provides for array operations on couplets of two dimensioned arrays of up to 32x32 elements. When in the array mode the addressing operators will load the arrays by column or by row into the proper stacks, perform the designated array operation placing the results in the third stack. Addressing operations will properly store the array result in accordance with the result array descriptor. Utilization of the array mode significantly reduces the code generated by the compiler for array operations.

The interrupt functions provided by the selected architecture provide for the setting of the priority level for each interrupt in the control memory. This will permit change in priority level of an interrupt function without engineering redesign. A masking function is provided to prevent the occurrance of selected interrupts during interrupt processing. Each of the interrupts causes transfer to an address of an interrupt array contained in the program control table. The compiler inserts the starting address of the appropriate interrupt processing routine into the interrupt array.

The architecture provides for the efficient implementation of the circumvention process. The machine states normally stored in the stack for subroutine entry and loop control are redundantly stored in dedicated locations. A recovery routine utilizes this data together with redundantly stored critical variables to reinitiate processing. The circumvention process is recursive, and permits the recovery routine to handle any number of interrupts.

The input/output functions for the selected architecture operate in a manner which is completely asynchronous of the main program execution. Input or output is initiated within the central processor function but then proceeds to completion without further program control. Information is passed between the memory of the central processor and the input/output on the basis of cycle stealing during an appropriate part of the instruction cycle. Input and output is initiated by addressing input or output descriptors in the program reference table. The descriptor contains sufficient information for the complete interpretation of the input or output function to be performed.

3.1.2 General System Description

3.1.2.1 General Organization

The Advanced Guidance Computer (AGC) is a general-purpose, stored-program machine whose design has been influenced by the physical limitations of its intended operating environment and by the requirement that most of the programs which it will execute will be generated by a compiler from source language statements written in a higher order language, namely SPL as specified in Part 1 of this report. The design of the AGC is aimed toward efficient object code representation of compiled SPL statements.

The AGC memory contains 32768 32-bit words of which 2048 words are scratch pad (NDRO) storage and 30720 words are electrically alterable (NDRO) storage. There is, in addition, a 1024-word high speed control memory which contains the microprogram used to implement the AGC instruction repertoire.

The logical organization of the AGC is composed of three principal sections:

Control: Fetches each AGC syllable while the current syllable is being executed. Syllables are fetched from the program memory one word at a time into a two-word register in a one-word look-ahead scheme that permits a two-byte syllable to span a word boundary. The control section then interprets the one or two-byte AGC syllables, furnishes the timing and directs the sequencing of events for the logical realization of programs.

Arithmetic: Carries out the arithmetic and logic functions as directed by the Control section.

Input/Output: Provides for the operation of input/output functions in a manner which is asynchronous to the main program execution. The main program initiates the input/output request and is interrupted when the request has been fulfilled.

3.1.2.2 Memory Allocation and Addressing

The Scratchpad storage section of memory is further divided into three subsections. These are:

Dedicated Register Area: These are memory locations used to hold registers and other control functions whose speed requirements or frequency of use do not appear to call for control registers or flip-flop implementation.

Scalar and Array Variable Storage: The function of this area is implied by its name and it immediately follows the dedicated register area. The first 1024 locations are available for the storage and retrieval of program alterable scalar variables. The first word of this 1024 sub-area is marked by the Scratchpad Memory (SPM) base address register. This base register is added to all ten-bit addresses of addressing syllables which refer to "scratch pad memory." Arrays which are accessed only by array descriptors may be stored anywhere in scratch pad memory and thus may be stored in the sub-area between scalar variable storage and the stack area (see below).

Scalar and Array Mode Stacks: This area is of a length sufficient to satisfy the maximum stack depth requirements of the operational guidance program and occupies the tail end of the scratchpad memory. Within it, operands are stored and fetched according to the address contained in the scalar mode pointer register rP. In array mode, whole arrays are loaded and stored and individual elements accessed according to the contents of the three array mode stack pointer registers P_0(=rP of scalar mode), P_1 and P_2.

Within the stack area of scratchpad memory, subroutine and function calling sequence arguments are located by the Calling Sequence Base Address Register (CSB). The current contents of CSB are preserved whenever a subroutine or function call is executed. The CSB is then reset to the highest stack address of a group of stack words consisting of calling sequence arguments and Return Parameter Words (RPW's).

The program storage section of memory is further divided into three subsections. These are:

Microprogram Storage Reserve: This is a 1024-word area which holds a copy of the microprogram operating out of a high-speed control memory. This copy is used to restore the latter whenever a radiation interrupt recovery is initiated.

Program Control Table (PCT): This is a 1024-word area whose first word is marked by the PCT base address register. The program control table is used to store descriptor words. These descriptor words define all variable and constant arrays, all subroutine and function entry points and all input/output operations.

General Program Area: The remaining approximately 28000 locations of program storage are available for storing the operational guidance program. Each program within this area is a subsection of or occupies one or more "program segments." Each program segment consists of up to 1024 program words and up to 1023 local data words. The first word of the program segment is marked by the Program Segment Base Address Register (PSB) and succeeding program words occupy successively higher addresses. The local data words of a program segment are also located relative to the setting of the PSB but the relative addressing is negative, i.e., successively higher local data as generated are stored in successively lower absolute addresses. Within the program area of the segment individual syllables may be addressed at the byte level while references to local single or double precision data conform to memory word boundaries.

3.1.2.3 Operating Modes and Control Stream Syllables

The AGC has two modes of operation:

Scalar Mode: Scalar mode handles all of the usual arithmetic, Boolean and logical computations that are performed on one or two-word data items including individual elements of arrays. The following types of program functions are restricted to scalar mode: input/output, transfer of control including subroutine or function calls, partial word operations.

Array Mode: Array mode allows for operations on whole one or two-dimensional arrays and array cross-sections (individual columns or rows of a two-dimensional array). The operations encompassed are simple assignment; array exchange; element-by-element addition, subtraction, multiplication and division; matrix multiplication including dot product; vector cross-product. The element-by-element operations provide for one of the operands to be a scalar. All array operations except simple assignment and exchange are elementary couplet operations in which a result must be stored each time and may not be retained without fetching as an operand for a further computation.

In either mode, AGC syllables are of four basic types which are identified by the first two bits of the instruction. These are:

Literal Fetch (00): Allows a 5 or 13-bit integer literal to be loaded into the arithmetic registers directly from the instruction itself without any additional memory access. The 5-bit literal is contained in a one-byte literal fetch syllable and the 13-bit literal in a two-byte literal fetch syllable. The literal fetch syllable is recognized in both array and scalar modes but has slight mode-dependent differences in effect.

Operators (01): Operator syllables are always one byte long and are used in scalar mode to perform arithmetic, Boolean, logical, manipulative, storage and control functions. The implied address(es) of the operand(s) is the one (are the two) item(s) at the top of the scalar mode stack. In array mode, operators are available to call out or facilitate the assignment, exchange, element-by-element, multiplication and cross-product operations. The implied addresses of the array operands are the array mode stacks. Two operators, EAM (enter array mode) and ESM (enter scalar mode), are recognized in either mode.

Addressing Syllables (10): Addressing syllables are two bytes long with the last ten bits containing an address relative to one of the four base registers. In scalar mode, addressing operators are used to fetch one or two-word operands to the top of the scalar mode stack or to store one or two-word results in scratch pad memory. In array mode the fetching operators are used to place scalar operands at the top of the array mode stacks. Such a fetch also causes the given array mode stack to be locked onto the scalar operand creating, in effect, an array all elements of which are identical.

Descriptor Call Syllables (11): Descriptor call syllables are also two bytes long with the last ten bits containing an address relative to the Program Control Table (PCT) or Calling Sequence Base (CSB). A descriptor call initiates the execution of an instruction which requires complicated supplementary data or elaboration of execution details that cannot be encoded into the descriptor call itself. This additional information is, therefore, encoded in a 32-bit word called a descriptor and the descriptor is located by the address field of the descriptor call. In scalar mode, descriptor calls are available for: fetching and storing an individual element of an array, the descriptor containing information on the precision, dimensions and base address of the array; calling a subroutine, the descriptor containing the address of the entry point and word count of the calling sequence; initiating input and output, the descriptor containing information on AD/DA conversion, channel identification, bit or byte transfer, serial or parallel operation. In array mode, descriptor calls are available for: fetching a whole array by row or by column; fetching an array cross-section; storing a whole array by column; storing an array cross-section; fetching a cross-product operand. In all of these cases the descriptor is an array descriptor as described for scalar mode element fetch and store.

Confining attention to scalar mode alone, the major categories of operators are:

Arithmetic: Nine basic arithmetic operators are provided including single and double precision floating point addition, subtraction, multiplication and division and integer quotient and remainder. In addition to the usual interrupts for floating point underflow and overflow, facilities are provided for controlling floating point operations on the basis of alignment and normalization shift limits.

Boolean: The six relational states (<, ≤, =; ≠, ≥, >) are represented by binary operators which produce the values "true" (1) or "false" (0) which can be combined by the operations for "logical or", "logical and", "logical exclusive or" and "not."

Decision and Branching: Operators are provided for unconditional jumps within and between program segments. Conditional branching instructions are based on the recognition of the values "true" and "false" as established by Boolean operations and also may be intra or intersegmental. A "universal fall through symbol" is recognized by most branching instructions to enable the execution of SPL declared switches with fall through positions.

Subroutines: Subroutine and function calls are accomplished by a descriptor call syllable as discussed above. Operators are provided for normal (immediately following point of call) and abnormal subroutine returns (to a label specified as an argument). The abnormal subroutine exit operator is capable of short-circuiting intervening levels of subroutine nesting to return to the level where the label argument is a local label. Calling sequence arguments and return control information are sequestered in the scalar mode stack and upon normal return this information is removed in a fashion that allows a functional result to remain on the top of the stack.

Indexing: Indexing in the conventional sense, that is creating effective addresses by index register modification, is implemental in a different manner in the AGC architecture. These architectural functions of indexing are met through the descriptor call syllables of scalar and array mode. Indexing on the AGC refers to provisions made for efficient compilation and execution of iterative loops (SPL FOR—loop). A single operator provides for the simultaneous setting of loop control registers and the preservation in the scalar mode stack of their previous contents together with the loop return point address. A single operator provides the facility for updating and testing the loop control parameter and for branching to the loop return point (iteration incomplete) or restoring the loop control registers to their previous values (iteration complete).

Partial Words: Extraction and insertion of signed or unsigned partial word fields is provided without the use of shift operations through the use of operators whose arguments are the starting bit and field length.

Storing: Although most needs for storing operands in scratch pad memory are met by the addressing syllables, several two-argument operators are provided to permit the storage of subroutine results where the location of the result is known to the subroutine as an address argument.

Intercept Processing/Circumvention: Operators are provided for accessing the interrupt condition and interrupt mask register. Interrupt conditions can, therefore, be enabled and disabled at the programmer's discretion during interrupt processing. An operator is provided to breakpoint the guidance program's operation at critical points by redundantly storing processor status information in dedicated locations. Another operation is provided to restore program operation to the last program point thus preserved.

3.1.2.4 Stack Concept and Function

The central feature which distinguishes the AGC from other guidance computer architectures is its organization about a last-in-first-out stack mechanism. The scalar mode stack consists of two arithmetic registers and the scalar mode stack pointer register (rP) which contains an address in the stack area of scratchpad storage. As operands are fetched from scratch pad memory, from the control stream, from the PCT, or from program storage, they enter the top-most arithmetic register. The previous contents of this register move to the lower arithmetic register and the previous contents of the latter move into the memory portion of the stack marked by rP. The stack mechanism is augmented by additional controls to designate the fullness/emptiness of an arithmetic register and a control which designates which of the two arithmetic registers is currently to be regarded as top-most. The net effect is a mechanism which achieves push-down pop-up capability in a manner that minimizes actual data movement and memory accesses.

Stack functioning is automatically adjusted by the presence of double precision operands so that storage of a double precision operand in the memory portion of the stack or its retrieval therefrom affects the two halves of the double-length arithmetic registers and two movements of the stack pointer register.

In the execution of operators, those which require a single operand first invoke a control subsequence which verifies the presence of at least one operand in the arithmetic registers and makes this register top-most if necessary; this accomplished, the operation proceeds. Operators that require two operands, first invoke a control subsequence which verifies that both arithmetic registers are filled, filling any empty register from memory and maintaining in the regis the reverse order of the operands' entry to the stack; this done, the operation proceeds. One-operand operations generally replace their arguments by their results. If we call the top-most arithmetic register rA and the other register rB, the pattern for a two-operand operation is rB op rA → rB, mark rA empty.

3.1.2.5 Mnemonic Descriptions

The definition of the AGC operator Mnemonics used throughout the remainder of this report are presented in Table 3-1. The following tables of instruction mnemonics present in alphabetical order all of the AGC instruction mnemonics. The instructions of both scalar and array modes are merged in this list with notation as to which mode they operate under. For each instruction the tabled date includes:

1. Mnemonic: This is generally a three-letter symbol. Certain arithmetic operators are represented by their conventional algebraic or SPL symbol with a qualifying letter, e.g., +S is single precision addition, ÷I is integer division.

2. Mode: A = Operates in array mode only.
   B = Operates in both array and scalar mode.
   S = Operates in scalar mode only.

3. Type: L = Literal fetch (1 or 2 bytes).
   O = Operator (1 byte).
   A = Addressing operator (2 bytes).
   D = Descriptor call (2 bytes).

4. Number of Operands: The number of operands required to be on the top of the stack (TOS). These operands are

accessible only when they occupy the two arithmetic registers at the TOS, rA and rB. Therefore, at the start of each operation a process of stack adjustment must be undertaken to make sure that at least one register is loaded for one operand operation or both are loaded for two operand operations. This preliminary stack adjustment is omitted from the functional descriptions of the instructions. If an instruction requires more than two operands, the step of adjustment required to get at the third, etc., operand is indicated by the symbol ADJ.

5. Functional Description: A brief formula and/or verbal description is given in terms of registers. Extremely complex operations such as subroutine entry (ESP) cannot be described briefly except in general functional terms.

6. Timing: A range of probable syllable execution times is given in microseconds (μs) for the scalar operations. These figures are based on the feasibility of utilizing a clock rate in the 5 to 10 mc range: Meaningful estimation of execution times for array functions can only be obtained by simulation, and will be heavily dependent on arithmetic/control unit hardware vs control memory/main memory tradeoffs and on array size.

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
+S	S	0	2	Single Precision Floating or Integer Addition. rB+rA→rB. Produces an integer result when both operands are integers if result has ≤24 bits. Floating point results are normalized and mantissa may have more than 24 bits.	2-4
-S	S	0	2	Single Precision Floating or Integer Subtraction. rB-rA→rB. See remarks pertaining to +S.	2-4
*S	S	0	2	Single Precision Floating or Integer Multiplication. rB*rA→rB. See remarks pertaining to +S.	14-18
÷RS	S	0	2	Single Precision Floating (Real) Division. rB÷rA→rB. Produces a floating point result without regard to operands.	16-20
+D	S	0	2	Double Precision Floating Addition. rB+rA→rB. Produces full 56-bit result from 56-bit operands.	4-6
-D	S	0	2	Double Precision Floating Addition. rB-rA→rB. Produces full 56-bit result from 56-bit operands.	4-6
*D	S	0	2	Double Precision Floating Multiplication. rB*rA→rB. Produces full 56-bit result from 56-bit operands.	16-20
÷RD	S	0	2	Double Precision Floating (Real) Division. rB÷rA→rB. Produces full 56-bit result from 56-bit operands.	18-22
÷I	S	0	2	Integer Division. 1. rA, rB if not already integers are integerized. 2. Truncated integer quotient [rB/rA]→rB. 3. Integer remainder →rA.	16-20
...	.	.	.	The following four operators are intended for use in procedures to store scalar results at addresses supplied as output parameters.	

Table 3-1

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
←C	S	0	2	1. rA contains an SPM relative a address; rB contains a single precision operand. 2. Store rB operand at address in rA. 3. Purge both address and operand from TOS.	2
⇐C	S	0	2	Same as ←C except rB contains a DP operand. Left half stored at address; right half at address +1.	2-4
←K	S	0	2	Same as ←C except only the address is purged.	2
⇐K	S	0	2	Same as ⇐C except only the address is purged.	2-4
ABS	S	0	1	Absolute Value Operator. Sign of rA or rB, whichever is at TOS, is set to 0.	1
ADE	S	D		Address of Element Descriptor Call.	
ADE,C				Upon inspecting array dimensions in the addressed array descriptor:	
			1	1. If array is nxl or lxn, one index (I) is present at TOS. The address of array element A(I) is left at TOS with I purged.	3-4
			or 2	2. Otherwise array is mxn with both m,n>l and two indices (I,J) are present at TOS (I beneath J). The address of element A(I,J) is left at TOS replacing both I and J.	3-4
AE+	A	0	2	Array Element Addition. Adds array or scalar in Stack 0 element-by-element to array or scalar in Stack 1 (not both scalars) and stores array result in Stack 2. The array dimensions and array vs. scalar character of the operands are set by the operand or array fetches used to load Stacks 0 and 1. The precision of the result is set by the array precision bit from the array descriptor.	

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
AE-	A	O	2	Array Element Subtraction. See remarks pertaining to AE+.	
AE*	A	O	2	Array Element Multiplication. See remarks pertaining to AE+.	
AE÷	A	O	2	Array Element Division. See remarks pertaining to AE+.	
AFC AFC,C	A	D	0	Array Fetch by Column. The base address, dimensions and precision bit of the addressed array descriptor are distributed to control registers and used to fetch the array by columns to Stack 0 or Stack 1 if Stack 0 already contains an array mode operand.	
AFR AFR,C	A	D	0	Array Fetch by Row. Same as AFC/AFC,C except that the array is fetched by rows.	
AND	S	O	2	Logical and Boolean And Operator. rB∩rA→rB. The result is 32 bits on a bit-by-bit basis.	1-2
ASC ASC,C	A	D	1	Array Store by Column. The array result stored by column in Stack 2 (see however operators SS0,SS1) is stored by column according to the addressed array descriptor. The dimensions of the result as determined by the preceding array operation (see AE+,AE-,AE*,AE÷,MXM and VXP) take precedence over the dimensions in the addressed descriptor.	
ASX	S	O	1	Abnormal Subroutine Exit. The operand at TOS is either a program reference descriptor (PRD) which specifies the location of an abnormal (alternate, error) return point from a subroutine or else it is an argument locator refering to the next most dynamic outer level of subroutine nesting. In the latter case, the operation iterates until the argument located by the argument locator is a PRD.	2-6
CHS	S	O	1	Change Sign (Unary Minus) Operator. The sign of rA or rB, whichever is TOS, is complemented.	1

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (µs)
CMP	S	O	1	Logical One's Complement Operator. The operand in rA or rB, whichever is TOS, is complemented on a 32-bit, bit-by-bit basis. The result replaces the argument. Note, this is not the Boolean Not Operator (NOT).	1
COX COX,C	A	D	1	Array Column Cross-Section Fetch. The operand at TOS is the column index and this is combined with data in the addressed array descriptor to fetch the column to Stack 0 or 1 as per AFC or AFR.	
DCI	S	D	0	Descriptor Call Indirect. The target of the CSB relative address is a descriptor call which is executed.	2-3
DUP	S	O	1	Duplicate Operator. rA is emptied if need be and a carbon copy of rB is propagated to rA.	1-2
EAM	B	O	0	Enter Array Mode. The ASMC flip-flop is set =1, and all instructions, most particularly descriptor calls, are given their array mode interpretation.	1
ELF ELF,C	S	D	1 or 2	Element Fetch Descriptor Call. Exactly like ADE/ADE,C except that the element rather than its address is fetched to the TOS.	4-6
EQ	S	O	2	Equal Relational Operator. If rA=rB then an integer 1 (0------01 = Boolean True) is left at TOS with both comparands purged. If rA≠rB, the comparands are replaced by 0 (=Boolean False).	1-2
EQUIV	S	O	2	Boolean Equivalence Operator. rB EQUIV rA+rB on a 32-bit, bit-by-bit basis: 1 EQUIV 1=1, 0 EQUIV o=1, o EQUIV 1=1 EQUIV 0=0.	1-2
ESC ESC,C	S	D	2 or 3	Element Store and Clear. Upon inspecting array dimensions in the addressed array descriptor: 1. If the array is nxl or lxn, the stack contents are an element A(I) beneath the index I. The element is stored and both element and index are purged.	2-3

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
				2. Otherwise array is mxn with both m,n>1 and stack contents are A(I,J) beneath J beneath I. The element is stored and all three operands are purged.	3-4
ESK ESK,C	S	D	2 or 3	Element Store and Keep. Same as ESC/ESC,C except that only the index or indices are purged.	2-3
ESM	B	0	0	Enter Scalar Mode. The ASMC flip-flop is set =0, and all instructions are given their scalar mode interpretation.	1
ESP ESP,C	S	D	0	Enter Sub-Program Descriptor Call. The contents of the arithmetic registers, if any, are pushed down into the memory portion of the stack. Then the contents of the PSB, CSB and PC registers (base addresses and location counter) together with a count of the number of words in the argument list (taken from the subroutine call descriptor addressed by the ESP/ESC,C instruction) are stored as two return parameter words (RPW_1, RPW_2) at TOS. These are then pushed down into the memory portion of the stack and PSB and PC are reset from the address field of the descriptor, while CSB is reset from the current contents of rP (scalar stack pointer register). Control is resumed at the location indicated in PC. Control is returned by the execution of an NSX operator to the byte following the ESP/ESP,C.	5-8
EXT	S	0	3	Partial Field Extract Operator. The topmost two operands in the stack are the field length ($1 \leq FL \leq 32$) and the starting bit ($0 \leq SB \leq 31$; $FL+SB \leq 32$; SB beneath FL). These are removed to special purpose registers. The third operand is then ADJusted and the specified field is extracted therefrom and left at TOS right-justified and left-zero-filled.	4-6

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (µs)
FCV	B	O	0	Fetch Current Value Register Operator. The contents of the CVR (current value register) are entered at TOS right-justified, left-zero-filled. The 14-bit CVR contains the value of the FOR-loop loop control variable at the innermost dynamic level of nesting. See the CR and NDX instructions.	2
FDC FDP FDS FDT	B	A	0	Fetch Double relative to C=CSB, P=PSB, S=SPM, T=PCT. The 10-bit address of these addressing operators is added to the appropriate base register and the contents of the two locations beginning at the resultant effective address are fetched to TOS as a double precision operand (left-hand half from lower address). The register precision flip-flop (APFF or BPFF) is turned on in scalar mode. In array mode the appropriate stack precision flip-flop (SPFF$_0$ or SPFF$_1$) is set as is the appropriate stack lock flip-flop (SLFF$_0$ or SLFF$_1$) thus marking the affected stack as containing a scalar operand.	4-6
FIC	S	O	0	Fetch Interrupt Condition Register. The contents of the 32-bit ICR are duplicated at the TOS. The ICR is unchanged.	2
FIM	S	O	0	Fetch Interrupt Mask Register. The contents of the 32-bit IMR are duplicated at the TOS. The IMR is unchanged.	2
FRAC	S	O	1	Fraction Operator. The integer bits of a single or double precision number are removed and the remaining fraction is normalized. If the operand was an integer, a zero result is generated. See INT.	1-3
FSR	A	O	1	Fetch Scalar Result. Transfers the single or double precision vector scalar (dot) product at the top of Stack 2 to top of Stack 0 making it available as a formula operand.	

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
FSC FSP FSS FST	B	A	0	Fetch Single relative to C=CSB, P=PSB, S=SPM, T=PCT. Operates like the corresponding FD-series shown above except that only one word of data is fetched (into the left-hand side of the receiving register). The register and stack precision flip-flop are turned off (=0). The array mode effect on the stack lock flip-flops is the same.	4-5
GQ GR	S	O	2	Greater Than or Equal Relational Operator. Greater Than Relational Operator. If $rB\left\{\begin{matrix}\geq\\>\end{matrix}\right\}rA$, the comparands are replaced by an integer 1(0--------01=Boolean True). Otherwise, the comparands are replaced by a 0(=Boolean False).	1-2
IN IN,C	S	D	0	Input Initiating Descriptor Call. Initiation of input begins with an interrupt of the central processor. After determining the type of input to be processed, the program executes an addressing descriptor call syllable. The descriptor call references the appropriate input file descriptor in the PRT, which is sent to the I/O for execution.	2-4
INS	S	O	4	Partial Field Insert Operator. The topmost two operands are as described under EXT. These are removed to special purpose registers whence they are used. The third operand is the receiver word and the fourth operand is the sender word. The last FL bits of the sender are shifted so that the first bit is in bit position SB. This field is inserted into the designated field of the receiver word with the other parts thereof unaffected. Only the receiver word remains at TOS.	6-8

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (µs)
INT	S	O	1	Integerize Operator. The floating point number at TOS is truncated to an integer as follows: Let e = the exponent, m = the mantissa, then 1. Do nothing if either m=o or e>o. 2. Replace TOS by 0 if e<-24. 3. If -24<e<o then shift mantissa right one bit and e←e+1. Repeat until e=o.	1-2
INX	S	O	1	Increment Index Operator. The increment value register is added to the current value register, i.e., IVR+CVR→CVR. Control is transferred to the 17-bit address (15 for word, 2 for byte) stored as the operand at TOS (last 17 bits). The operand is not purged from the stack.	2-3
JMP	S	O	1	Jump Operator. 1. If operand is <o it is a program reference descriptor. Operation is same as JOS, q.v. 2. If operand is >0 but field (1//19) ≠0, continue operation in sequence. 3. Else interpret last 12 bits as jump address as follows: Word = PSB+first 10 bits. Byte = last 2 bits.	2-4
JOF	S	O	2	Jump on False Operator. rA contains an operand which is interpreted exactly as in the JMP instruction save only that the jump is conditioned on the contents of rB. If the last bit of rB (bit 31) is 0, the jump is taken, otherwise control continues in sequence. Both operands are purged jump or no jump.	1-4
JOS	S	D	0	Jump out of Segment Descriptor Call. The PCT-relative addressed descriptor is a program reference descriptor (PRD) which contains a segment base address (→PSB) and an entry point displacement (epd; Word = first 10 bits of epd + PSB Byte = last 2 bits of epd) Control is transferred to the word and byte as thus determined.	2-4

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
JOT	S	O	2	Jump on True Operator. Logical dual of JOF. Last bit of rB must =1 for jump to take place.	1-4
LFS	B	L	0	Literal Fetch Syllable. If third bit of this instruction is =0, then instruction is 1-byte long. Last 5 bits of instruction are sent to TOS, right-justified and left-zero-filled. If third bit =1 then instruction is 2 bytes long. Last 13 bits of instruction are sent to TOS, right-justified and left-zero-filled. Sets $SLFF_0$ or $SLFF_1$ in array mode.	1
LQ LS	S	O	2	Less Than or Equal Relational Operator. Less Than Relational Operator. Exactly like GR/GQ as regards the relations $\begin{Bmatrix}\leq\\<\end{Bmatrix}$	1-2
MAX MIN	S	O	2	Maximum Operator: Max $\{rA, rB\}\rightarrow rB$. Minimum Operator: Min $\{rA, rB\}\rightarrow rB$. The rejected comparand is purged.	
MXM	A	O	2	Matrix Multiply. The operands are both arrays in Stacks 0 and 1 with matching inner dimensions. The product is generated by columns and stored in Stack 2 whence it may be stored in memory by an ASC/ASC,C descriptor call or sent to top of Stack 0 by an FSR operator if the result is 1x1.	
NDX	S	O	2	Index and Test Operator. The loop control registers are incremented and tested 1. IVR+CVR→CVR 2. <u>If</u> IVR>0 and CVR<FVR or IVR<0 and CVR>FVR <u>then</u> control is transferred to the address stored at TOS (See INX instruction) <u>else</u> the address operand is purged and the index control word (ICW) is used to reset the loop control registers as follows:	2-4

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
				field of ICW→FVR field of ICW→IVR field of ICW→ACVR and reset CVR with contents of address now stored in ACVR.	
NIX	S	O	1	Purge Operator: The operand at TOS is purged.	1-2
NOP	S	O	0	No Operation: Advance to next sequential instruction.	1
NOT	S	O	1	Boolean Not Operator: The result is 31-zeroes followed by the complement of the last bit of the argument. NOT applied to any argument results in the truth/falseness of itseven parity.	1-2
NQ	S	O	2	Not Equal Relational Operator: Same as EQ vis-a-vis ≠.	1-2
NSX	S	O	0	Normal Subroutine Exit: Use the current value of the CSB to locate the RPW's. Use the contents of these to reset the PSB,PC,rP and CSB. Control resumes at the byte following the ESP/ESP,C which caused the RPW's to be stored. Note that rP is wound down to the point it was at before the first argument of the calling sequence was stored or generated in the stack. When the stack is thus wound down, the arithmetic registers are not affected, providing a place for the return of function sub-program results.	6-8
OR	S	O	2	Logical Boolean Or Operator. rB∪rA→rB. The result is 32 bits on a bit-by-bit basis.	1-2
OUT, OUT,C	S	D	0	Initiate Output Descriptor Call: Output is initiated by the central processor either as a result of completing some program function or by interrupt. The processor executes a descriptor call syllable. The descriptor call references the appropriate output file descriptor in the PRT, which is sent to the I/O for execution.	2-4

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (µs)
PCR	S	O	0	Program Recover Return: Returns processor to state recorded by last executed PRR. Used for radiation circumvention recovery.	
PCV	S	O	0	Preserve Current Value Register: The CVR is stored at the address in the ACVR.	2
PRR	S	O	0	Program Record: Redundantly stores processor status information in dedicated area. Permits roll back to recorded point after radiation interrupt.	6-8
RND	S	O	1	Round Operator: If TOS contains a single precision operand (APFF or BPFF=0) then the first extension bit (25th mantissa bit) is added to the 24th mantissa bit and exponent is adjusted if necessary. The extra mantissa bits can result from all of the single precision arithmetic operations and from INT. If TOS contains a double precision operand (APFF or BPFF=1) the same rounding operation takes place and APFF or BPFF is set to 0.	2-3
ROX RØX,C	A	D	1	Array Row Cross-Section Fetch: The operand at TOS is the row index and this is combined with data in the addressed array descriptor to fetch the row to Stack 0 or 1 as per AFC or AFR.	
SCV	S	O	1	Set Current Value Register: Last 14 bits of operand at TOS are stored in the CVR. The operand is purged.	1-2
SCX SCX,C	A	D	1	Store Array Column Cross-Section: The topmost value on Stack 2 is taken to be the column index. The column vector reposing in Stack 2 beneath this index is stored at the appropriate part of the array whose descriptor is addressed.	

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
SDC	S	A	1	Store Double and Clear: If the operand at TOS is single precision (APFF or BPFF=0), the right half of the appropriate register (rA or rB) is zeroed and the operation proceeds as double precision. If a double precision operand is present at TOS, the left half is stored at the given address (relative to SPM) and the right half at the next higher location. The operand is purged.	2-3
SDK	S	A	1	Store Double and Keep: Same as SDC except that the operand is not purged.	2-3
SDZ	S	A	0	Store Double Zero: A double precision zero (two words of all zero) is stored at the given address (relative to SPM) and the next higher location. The stack is not affected.	2-3
SGN	S	O	1	Signum Operation: If the operand at TOS is <0 it is replaced by an integer -1. Otherwise, it is replaced by an integer +1.	1
SIC	S	O	1	Set Interrupt Condition Register: The operand at TOS is stored in the 32-bit ICR. The operand is purged.	1-2
SIM	S	O	1	Set Interrupt Mask Register. The operand at TOS is stored in the 32-bit IMP. The operand is purged.	1-2
SNS	S	O	4	Signed Partial Field Insert Operator: Same as INS except that before the field "last FL bits of the sender" is moved into position for insertion, this field is reconstituted by circularly shifting the sign bit of the sender (bit 0) into the last bit position of the sender (bit 31).	8-12
SRC	S	A	1	Store Rounded and Clear: Performs actions of RND on operand at TOS and then stores the single precision result at the given (SPM relative) address. The operand is purged.	2-3
SRK	S	A	1	Store Rounded and Keep: Same as SRC except that rounded operand is not purged.	2-3

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (μs)
SRX SRX,C	A	D	1	Store Array Row Cross-Section: Same as SCX/SCX,C vis-a-vis row cross-section.	
SSO SS1	A	O	0	Set Store from Stack $\left\{\begin{matrix}0\\1\end{matrix}\right\}$. These operations allow simple array assignment and exchange statements to be executed without the normal "load Stack 0, load Stack 1, Perform operation, Unload Stack 2" sequence.	
SSZ	S	A	0	Store Single Zero: A zero word is stored at the given (SPM relative) address. The stack is not affected.	2
STC	S	A	1	Store Truncated and Clear: Same as SRC except the preliminary RND is replaced by a TRC.	2-3
STK	S	A	1	Store Truncated and Keep: Same as SRK except the preliminary RND is replaced by a TRC.	2-3
SXT	S	O	3	Signed Partial Field Extract Operator: Exactly the same as EXT except at the very end, bit 31 of the result is rotated into the sign bit position.	6-8
TRC	S	O	1	Truncate Operator: Zero out the right half of the double length TOS register and turn off the appropriate precision flip-flop if it is on.	1-2
VXP	A	O	2	Vector Cross Product: The vector cross product of the 3-vectors in Stacks 0 and 1 is developed and stored in Stack 2. The operands are actually fetched twice into each of the stacks. See XPF.	
XCH	S	O	2	Exchange Operator: The contents of rA and rB are exchanged: rA⇄rB. This is actually done without moving the data by complementing the c-bit which specifies the identity of the arithmetic registers.	1

Mnemonic	Mode	Type	# Ops.	Functional Description	Timing (µs)
XCR	S	0	4	Index Create Operator: The four operands are, from the top down - the commencing value the final value the increment value and the SPM relative address of the variable whose value is at TOS. These refer to the parameters of a FOR-loop and the address of the loop control variable. 1. The current contents of IVR, FVR and ACVR are packed and stored as an index control word (ICW) in the stack. The CVR is stored at the address in the ACVR. 2. The absolute address of the byte following the XCR is stored at TOS as a loop return point (LRP). 3. The four operands are distributed to CVR, FVR, IVR and ACVR in top down order and are purged. (The stack additions of steps 1 and 2 actually take place after step 3).	8-12
XOR	S	0	2	Boolean and Logical Exclusive Or Operator: The 32-bit, bit-by-bit exclusive or (symmetric difference) of rA and rB is developed and stored in rB. rA is purged.	1-2
XPF XPF,C	A	D	0	Cross Product Operand Fetch: The 3-vector whose descriptor is addressed is loaded twice to Stack 0 or Stack 1 (if Stack 0 contains an operand).	

3.2 Arithmetic and Control Function Summary

The following is a summary of the arithmetic and control functions of the AGC. The summary is broken down by functional category and each category is accompanied by a brief description of its function. Those functions whose implementation is presently planned to be a hardware register or flip-flop are starred (*). Those functions which are not starred are to be represented as fields in control memory. Some of the functions are listed under more than one category.

3.2.1 Arithmetic and Stack Control

Name		No. Bits	Function
P	*	11	Stack pointer register. Gives the scratch pad memory address of the top word of the memory portion of the arithmetic stack.
AR_0	*	64	Double length arithmetic registers representing the arithmetic register portion of the arithmetic stack.
AR_1	*	64	
F_0		1	Arithmetic register validity flip-flops. If off, the corresponding AR is regarded as empty.
F_1		1	
PFF_0		1	Arithmetic register precision flip-flops. If $\begin{vmatrix}\text{on}\\\text{off}\end{vmatrix}$, the corresponding AR holds a $\begin{vmatrix}\text{double}\\\text{single}\end{vmatrix}$ precision quantity.
PFF_1		1	
c		1	Top register flip-flop.

When ⟶	c=0	c=1
The Control ↓	is called	is called
AR_0	rA	rB
AR_1	rB	rA
F_0	a	b
F_1	b	a
PFF_0	APFF	BPFF
PFF_1	BPFF	APFF

3.2.2 Instruction Fetching and Decoding

Name		No. Bits	Function
IR	*	64	Instruction Register: Double-length, wrap-around register for fetching instruction words with one word look-ahead.
CS	*	16	Current Syllable Register: Holds the 1 or 2 byte syllable gated from IR, currently being executed.
CB	*	3	Current Byte Register: Holds the number (0th to 7th of IR) of leading byte of syllable in CS.
NB	*	3	Next Byte Register: Holds the number (0th to 7th of IR) of leading byte of syllable which will follow syllable in CS (if a jump does not occur).
NS	*	15	Next Syllable Register: Holds the word address of the word in IR which contains byte NB.
DDR		32	Descriptor Decoding Register: Holds for decoding a descriptor fetched by a descriptor call syllable.
ILFF	*	1	Instruction Length Flip-Flop. = 0 if current syllable is 1 byte long = 1 if current syllable is 2 bytes long

3.2.3 Addressing

Name		No. Bits	Function
CSB	*	11	Calling Sequence Base Register: Base address of words in current level procedure calling sequence.
PSB	*	15	Program Segment Base Register: Base address of current program segment.
SPM		11	Scratch Pad Memory Register: Base address of writable portion of memory.
PCT		15	Program Control Table Register: Base address of program control table portion of NDRO memory.

Partial Word Operations

rS	*	5	Starting Bit Register: Holds bit number (0 left to 31 right) of starting bit of field for extraction or insertion.
rL	*	5	Field Length Register: Holds the length minus 1 of said field.

These two registers are coincident with array mode registers rM_0, rN_0 respectively.

3.2.4 Index Operations

Name	No. Bits	Function
ACVR	10	Address-of-Current-Value Register: Holds address relative to SPM of the loop control variable at the dynamically innermost level of nesting.
CVR *	14	Current Value Register: Contains current value of the loop control variable defined under ACVR.
FVR *	14	Final Value Register: Contains the $\begin{vmatrix}\text{lower}\\ \text{upper}\end{vmatrix}$ limit on said loop control variable as IVR (below) is $\begin{vmatrix}<0\\ >0\end{vmatrix}$.
IVR *	8	Increment Vaue Register: Contains the seven-bit plus sign increment that will be applied to CVR by CVR = CVR + IVR at the end of the loop.

3.2.5 Floating Point Control

Name		No. Bits	Function
ASL	*	6	Alignment Shift Limit Register: Holds limit value on number of bits mantissa is shifted to align exponents. If value in ASL is exceeded, the
ASFF		1	Alignment Shift Flip-Flop is set.
NSL	*	6	Normalization Shift Limit Register: Holds limit value on number of bits mantissa of result is shifted to normalize it. If value in NSL is exceeded, the
NSFF		1	Normalization Shift Flip-Flop is set.

3.2.6 Interrupts

Name		No. Bits	Function
IFF		1	Interrupt Flip-Flop: Is set when any interrupt is detected and allowed.
ICR	*	32	Interrupt Condition Register: Holds 32 indicators each specific to a different interrupt condition.
IMR	*	32	Interrupt Mask Register: Holds 32 enable/disable indicators (0 enable, 1 disable) corresponding one-to-one with ICR.
IPR	*	4	Interrupt Priority Register: Holds the priority level (0-15) of the interrupt currently being processed.
PFW		128 (4 words)	Priority Field Words: These are 32 four-bit fields each containing the priority of one of the 32 possible interrupt conditions.

3.2.7 Array Processing

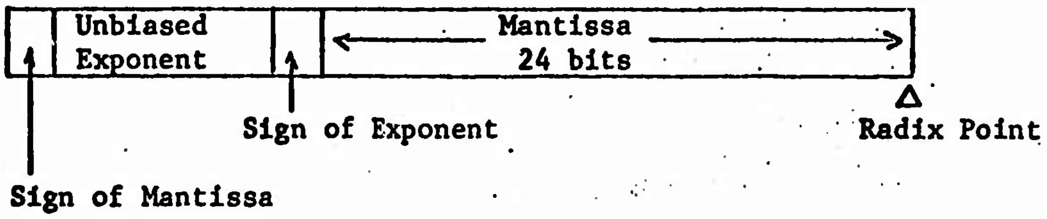
Name		No. Bits		Function
ASMC		1		Array/Scalar Mode Control: Indicates whether the processor is in scalar mode (=0) or in array mode (=1). This flip-flop is not specifically mentioned in the text of the report.
P_0	*	11		Stack Pointer Registers for the three array mode stacks. P_0 is P of the scalar mode arithmetic stack.
P_1	*	11		
P_2	*	11		
ARR	*	64		Array Result Register: Auxilary arithmetic register for holding intermediate and final element result in array computations.
rM_i	*	5	i=0, 1	Number of Rows
rN_i	*	5		Number of Columns
BA_i		15		Base Address of Array
$SLFF_i$		1		Stack Lock Flip-Flop (=1 when Stack i holds a scalar operand)
$SPFF_i$		1		Stack Precision Flip-Flop (=1 when Stack i holds a double precision scalar or array)

3.3 Data Representation

3.3.1 Numeric

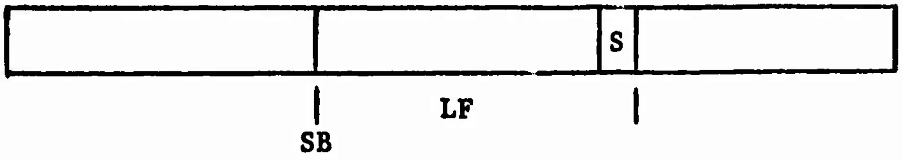
Numeric data is represented on the AGC either as (single precision) integers or single precision or double precision floating point numbers.

A single precision floating point number is represented by a 24-bit mantissa plus sign with a six-bit plus sign (unbiased) base-two exponent. The mantissa is normalized whenever the exponent is non-zero and the radix point is located at the right-hand-end of the mantissa. A double precision floating point number has the second word as an unsigned continuation of the mantissa.

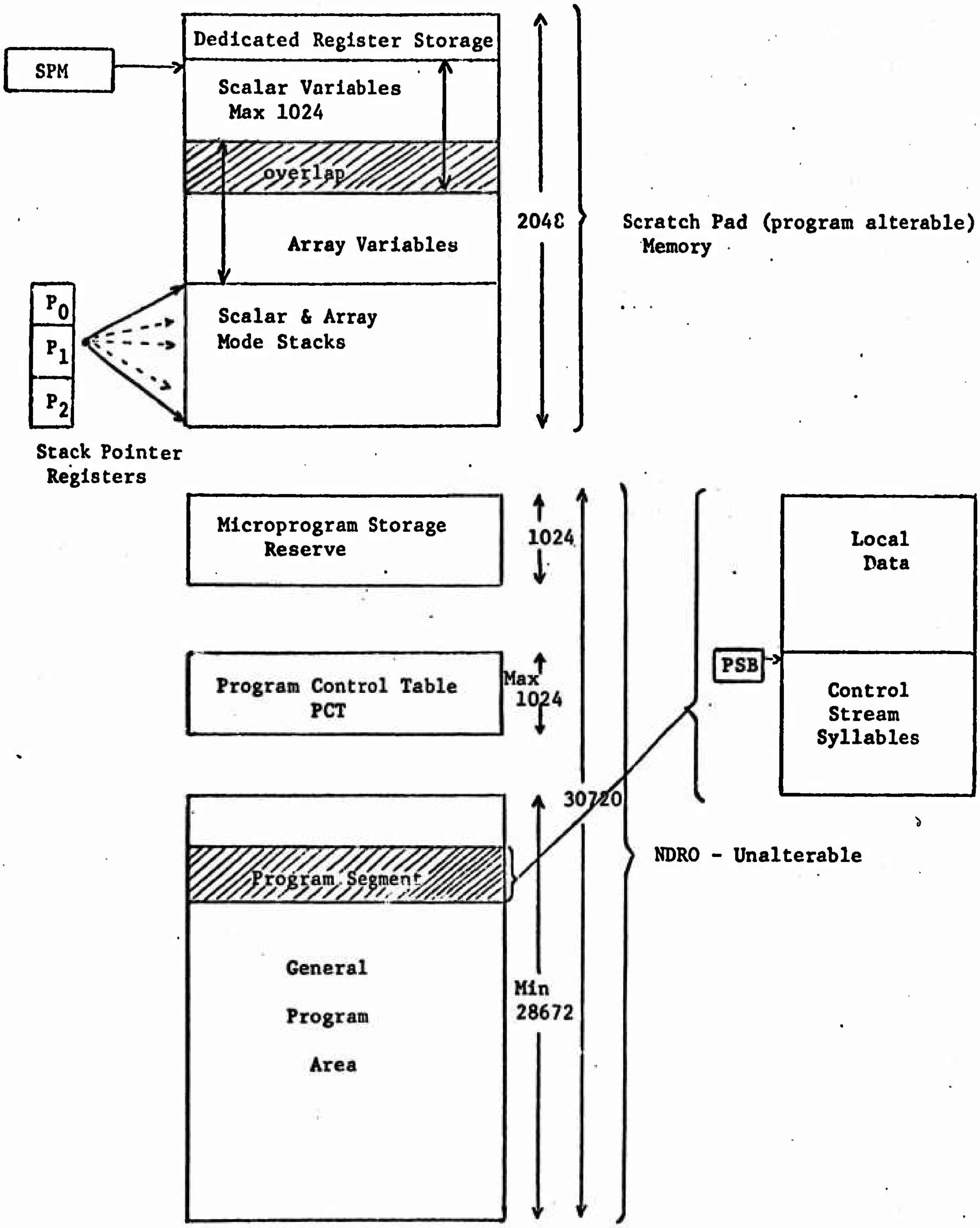


3.3.2 Partial Word

A partial word is represented in the AGC as a field within a word (may be the entire word) which begins within the word at a starting bit (SB) and whose length in bits is within the word (LF). The field may be signed. If signed, the sign is carried at the right hand end of the field and is considered part of LF.



3.4 Memory Allocation



3.5 Program String Syllables

Syllable Type	Length (bytes)	First Two Bits	Purpose
Literal Fetch	1 or 2	00	Bring integers $\leq$ 8191 into TOS.
Operator	1	01	Perform no-address operations.
Addressing Operator	2	10	Address operations. Address operands directly.
Descriptor Call	2	11	Address operations. Address arrays, subroutines, etc., indirectly via descriptor words.

3.5.1 Literal Fetch Syllable (LFS)

An LFS is used to load a positive integral value to the top of the stack.

0 0 0	X X X X X	for literals $\leq$ 31
0 0 1	X X X X X X X X X X X X X	for literals $\leq$ 8191

3.5.2 Operator Syllable

The format of an operator syllable is

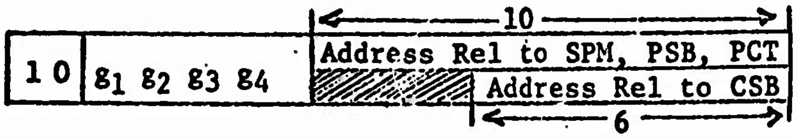
0 1	f f f f f f

where f is specific to the operation to be performed and is used for such functions as +, -, ÷, AND, OR, etc.

3.5.3 Addressing Operator Syllable

An addressing operator is always two bytes long with the last six or ten bits constituting an address relative to one of the four base address registers SPM (Scratch Pad Memory), PSB (Program Segment Base), PCT (Program Control Table) and CSB (Calling Sequence Base).

The format of an addressing operator is



where g is specific to the function to be performed , such as

FSS (fetch a single precision operand relative to SPM),

SRK (round-off and store a single precision operand and keep it for further use).

The function of an addressing operator is to fetch to or store from the TOS a data item where the address refers directly to the item.

3.5.4 Descriptor Call Syllable

A descriptor call syllable is two bytes long with the same address structure as an addressing operator except that the address pertains only to the PCT (ten bits) or the CSB (six bits). The entity referenced by the address is called a descriptor.

The format of a descriptor call syllable is

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td></tr>
<tr><td rowspan="2">1</td><td rowspan="2">1</td><td>0</td><td rowspan="2" colspan="3">h h h</td><td colspan="10">Address Relative to PCT</td></tr>
<tr><td>1</td><td colspan="4">▨</td><td colspan="6">Addr Rel to CSB</td></tr>
</table>

where h is specific to the function to be performed, and bit 2 determines whether the address is relative to the PCT or the CSB for $0 \leq h \leq 5$.

The h-bits select such functions as:

ESP (enter sub-program - the descriptor contains the absolute address of the entry point and the number of words in the calling sequence),

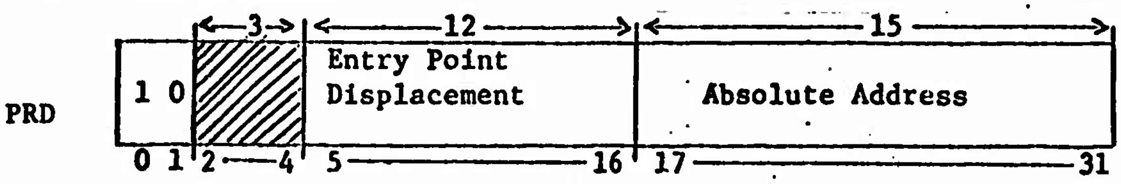
ELF (element fetch - the one or two indices are at the TOS - the descriptor contains the base address, dimensions and precision [single vs double] of the array).

3.6 Descriptor Summary

3.6.1 Program Reference Descriptor (PRD)

The Program Reference Descriptor permits entry to a program segment at its beginning (base address) or to any syllable within the segment (entry point displacement).

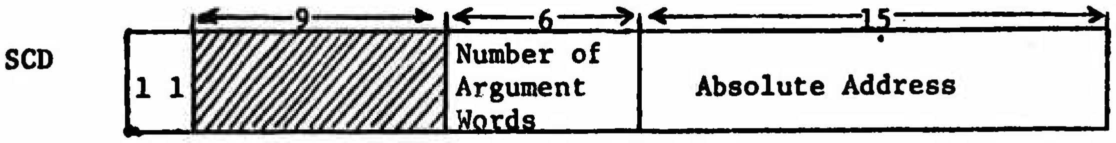
The Program Reference Descriptor in the Program Control Table (PCT) has the following format:



The entry point displacement (epd) consists of 10 bits of word address and 2 bits of byte address. The word part is added to the absolute address.

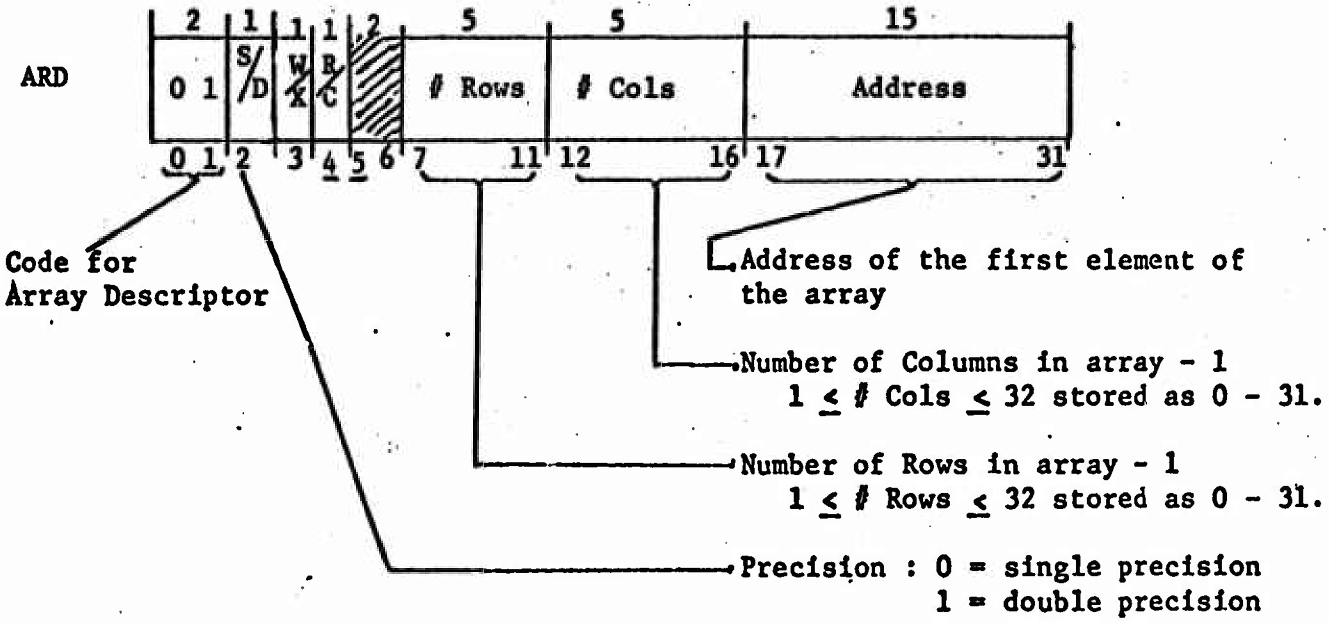
3.6.2 Subroutine Call Descriptor (SCD)

The subroutine call descriptor permits entry to a subroutine, a procedure or a function. The SCD has the following format:



3.6.3 Array Descriptor (ARD)

Array Descriptors in the PRD describe the location and structure of data arrays.. They are used in the scalar mode for accessing an array element, and in the array mode for a variety of array processing functions. An array descriptor has the following format:


ARD
2
1
1
1
2
5
5
15
0 1
S/D
W/X
R/C
# Rows
# Cols
Address
0 1
2
3
4
5
6
7
11
12
16
17
31
Code for
Array Descriptor
Address of the first element of the array
Number of Columns in array - 1
1 ≤ # Cols ≤ 32 stored as 0 - 31.
Number of Rows in array - 1
1 ≤ # Rows ≤ 32 stored as 0 - 31.
Precision : 0 = single precision
1 = double precision

Bit 3: W/X = 0/1 for Whole Array vs. Cross-Section.

Bit 4: R/C = 0/1 for Row vs Column Cross-Section if Bit 3 = 1

These two bits are not part of an ARD as stored in the PCT, but are created at object time by compiler supplied code and placed in the stack when an array cross-section is to be used as an argument to a subroutine. The address is also altered to the address of the leading element of the cross-section. Thus, the argument generated for A(,J) is

```
   (ARD(A) + J* (ARD(A)(7//5) + 1)*(ARD(A)(1//2) + 1)) LOR

                                              HEX'18000000'
```

3.7 File Descriptor (FD)

In the AGC architecture all input/output is viewed as data files. File Descriptors in the PCT describe the characteristics and location of input and output data. File Descriptors are accessed by descriptor call syllables, and are sent to Input/Output Control. Once initiated Input/Output proceeds in an asynchronous manner to completion, which is indicated by setting interrupt conditions. The decoupling of the Input and Output from the AGC CPU functions is achieved by providing separate input and output buffer registers for both serial and parallel input and output data. The format of the File Descriptor is:

FD	0 0	File Control	Record Control	Memory Address
	0 1	2 11	12 21-	22 31

Code for File Descriptor

The file control field is used to pass control information to the I/O section of the AGC architecture. It is issued to specify such things as device and chained information and to indicate whether the file is composed of single or multiple word data.

The record control field is used to indicate such things as the location of record data within a word, and the significance of data content in relation to input or output discretes.

The memory address represents the beginning address in scratchpad memory where the first word of the input will be placed, or the first word of output will be sought. If the input data is less than a word (a field) then the data will be contained within that word.

4.0 Architecture Description

4.1 Memory Structure & Addressing

The current AGC architecture memory addresses 32768 words of 32-bits each of which the first 2048 are alterable (scratch pad memory) and the last 30720 are unalterable (or NDRO). A memory map is shown in Figure 4.1.

4.1.1 Scratch Pad Memory

The addressing structure further subdivides scratch pad memory into variable length portions for the scalar and array mode stacks, scalar variable storage, array variable storage and dedicated register storage.

4.1.2 Stack Storage & Addressing

The stack storage area is addressed by the stack pointer registers P_0, P_1 and P_2 of which only the first is active in the scalar mode. The scalar stack pointer register P ($\equiv P_0$) links the arithmetic registers rA and rB to memory to achieve a LIFO stack mechanism (see Section 4.4). In array mode, the pointers P_0, P_1, P_2 serve as memory-to-memory links for loading and storing whole arrays (see Section 4.10). Each of the registers P_0, P_1, P_2 is 11 bits in length and represents a direct address, i.e., an address which does not have to have a base address (register) added in to arrive at an effective address. The stack area is at the high end of the scratch pad memory.

Within the scalar mode stack, data may be addressed relative to the calling sequence base (CSB) register. This is discussed at length in Section 4.9.

4.1.3 Dedicated Register Storage

Dedicated register storage is at the low end of the scratch pad memory. This area is of fixed length and contains all registers, flip-flops, etc., whose frequency of use and speed requirements do not necessitate direct hardware implementation.

Some of the entities in this area are made available for testing or manipulation by the program via the SPL HARDWARE declaration.

4.1.4 Scalar Variable Storage

Scalar variables are one or two word items which are fetched by the addressing operators FSS or FDS, stored by any of eight storage addressing operators or stored by any of four operators which find the address as well as the data to be stored as the first two elements in the top of the stack. In all these operations, the addressing operation spans ten bits or 1024 words. The addresses in these operations are interpreted as being relative to the SPM (scratch pad memory register).

The SPL program refers to scalar variables as unsubscripted identifiers which have been typed as integer, floating, Boolean, status or logical variables without the CONSTANT attribute.

4.1.5 Array Variable Storage

One and two-dimensional arrays which are to be altered during program execution are allocated by the compiler to the upper portion of the scratch pad area where some of the addresses may require eleven bits. This does not require a base address (like SPM) since all arrays are addressed via array descriptors. These are 32-bit words which contain the full 15-bit base address of the array as well as array dimensions and storage mode (single or double precision). The array variable storage area may overlap the scalar variable storage area depending on the structure of the particular program.

4.1.6 NDRO - Unalterable Memory

This section of memory cannot be altered by the operating program. This memory section is further divided into areas for microprogram storage reserve, program control table and general program area.

4.1.7 Microprogram Storage Reserve

It is anticipated that the logical control of the AGC will be implemented via microprogramming. The microprogramming infrastructure will be heavily slanted toward the stack concept which AGC uses. It is further anticipated that the microprogram storage will be equivalent to 1024 x 32-bits of very high speed memory (not shown in Figure 4-1). When AGC operation is resumed after a radiation interrupt, the first task that must be accomplished is to restore the microprogram high speed memory from the microprogram storage reserve held in NDRO.

4.1.8 Program Control Table

The program control table or PCT is a variable length section of maximum length 1024. It is intended principally as a depository for descriptors which control the program flow and describe complex data. All scalar and array mode references to arrays, subroutine calls, intersegmental jumps, and I/O are done via indirect addressing, i.e., the control stream syllable (descriptor call) which has a ten-bit address fetches one of the 1024 PCT words to the descriptor decoding register (DDR) where it is broken up and distributed to other control registers.

In addition to descriptors, globally defined scalar constants may be stored in the PCT, e.g., π, $\pi/2$, degrees ⇆ radians, earth's polar and equatorial mean diameters, etc. These may be feteched by the PCT relative addressing operators

FST, FDT. Constant arrays of global interest may also be stored in the PCT, e.g., launch and aim point coordinate vectors. These are addressed (for fetching purposes only) with descriptor calls to the appropriate array descriptors.

Scalar and array constants arise in SPL programs from giving the CONSTANT attribute to named data entities. Globality is conveyed either by use of a COMPOOL or by placing data declarations in the outermost (main program) level of a nested block.

4.1.9 General Program Area

The general program area contains data and control stream syllables (i.e., instructions) organized into program segments and subroutines (which are also program segments). Each program segment contains in addition to its control stream those constants and literals which are purely local to itself. Integer (including Boolean) literals which are less than 8191 in absolute value are embedded in the control stream itself and are fetched by the literal fetch syllable. Integer literals greater in magnitude than 8191 and all other non-integer literals and constants are stored backward relative to the beginning of the program segment which is marked by the program segment base (PSB) register. These are then accessible via the FSP and FDP addressing operators.

These matters are more fully described in Sections 4.5, 4.8 and 4.10.

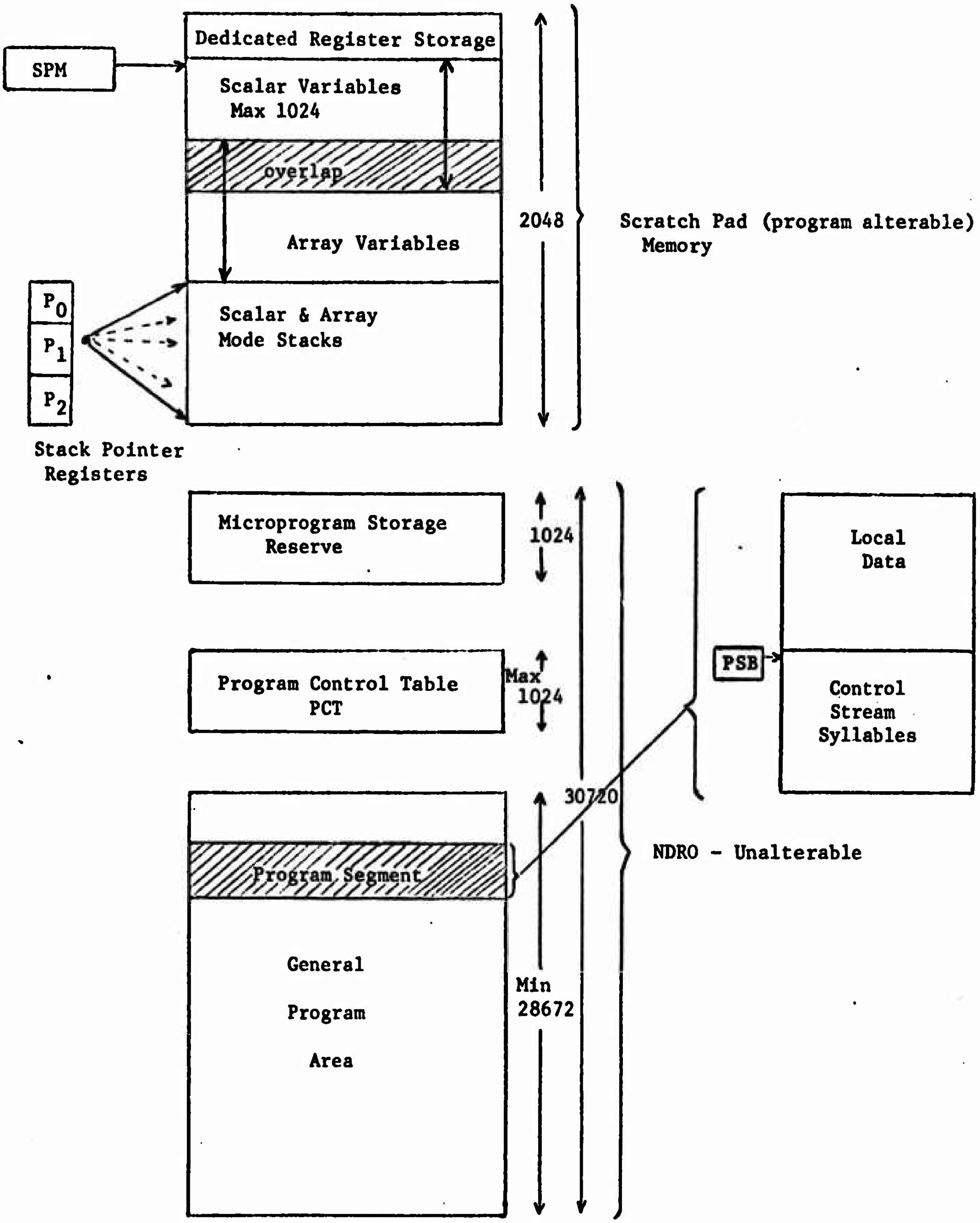


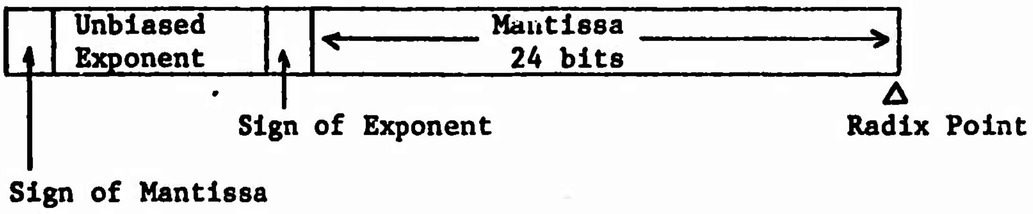
Figure 4-1 : Memory Map

4.2 Numeric Data & Arithmetic

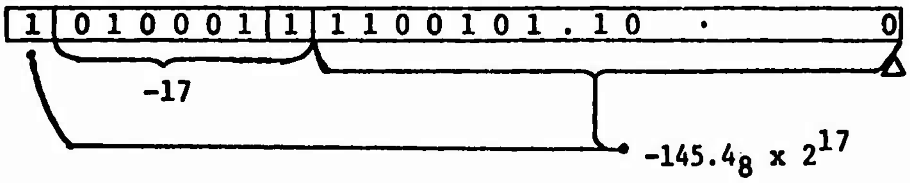
4.2.1 Formats

Numeric data is represented on the AGC either as (single precision) integers or single precision or double precision floating point numbers.

A single precision floating point number is represented by a 24-bit mantissa plus sign with a six-bit plus sign (unbiased) base-two exponent. The mantissa is normalized whenever the exponent is non-zero and the radix point is located at the right-hand-end of the mantissa.



Example: $-101.5_{10} = -145.4_8$



i.e., $-145.4_8 = (-145.4_8 \cdot 2^{17}) \times 2^{-17}$

mantissa exponent

The position of the exponent sign at the end of the exponent makes the exponent compatible with partial word operations (q.v.). The position of the radix point at the right-hand-end of the mantissa has two consequences:

- first, the range of numerical magnitudes representable in this format is not symmetric about zero, i.e.,

  magnitude smallest number = $2^{23} \times 2^{-63} = 2^{-40}$

  magnitude largest number = $(2^{24}-1) \times 2^{63} < 2^{87}$

- second, integers can be represented as right justified (unnormalized) floating point numbers with exponent-zero. This means that floating point numbers and integers may be freely intermixed in a computation without the need to convert from one format to another. A whole number may thus have two representations: as a normalized floating point number or as an integer

5 =	+	0	+	0 —— 0 5	integer
5.0 =	+	25_8	-	5 0 —— 0	flt. pt.

The latter fact also has consequences for the compactness of the control stream. Small integer literals ($\leq$ 8191) may be fetched out of the control stream for use in floating point computations (LFS syllable, Section 4.5).

A double precision floating point number is simply a two-word floating point numbers in which the second word is an unsigned continuation of the mantissa.

Example: $0.6 = 0.\overline{\dot{4}631}_8$ † =

+	30_8	-	46314631	46314631462

$\longleftarrow$ word 1 $\longrightarrow$ | $\longleftarrow$ word 2 $\longrightarrow$

last digit
base 4

4.2.2 Arithmetic Operations

The arithmetic operations +, -, * and ÷R are available in single and double precision versions. If we adopt a ranking of types from low to high

I (integer) < S (single) < D (double)

then any operation between two of these takes the type of its highest type operand. If both operands are integers (exponent = 0) then an integer result is produced unless the mantissa would exceed 24 bits. The exception to this

† Notation $.\overline{\dot{a}bc}$ means .abcabc ... Thus, 1/7 = $0.\overline{\dot{1}42857}$ (base ten)

rule is division ÷R which produces a floating point result even when both operands are integers. It should be noted that single precision operations generally produce double precision results which may be retained at the higher precision, truncated or rounded.

The integer divide operation ÷i is for single precision operands only. If the operands are not already integers, they are made so by truncation before the division is performed. The operation produces the truncated integer quotient <u>and</u> the integer remainder.

<u>Example</u>: Suppose X = 101.5, Y = 10.3

X ÷ Y = 101 ÷i 10 = (10, 1)

where 10 is the truncated quotient

and 1 is the remainder.

4.2.3 Control of Floating Point Operations

The AGC provides a degree of control over floating point operations not usually found in even large general purpose computers. This control is exercised through alignment and normalization control in addition to the usual trapping of floating point overflow and underflow.

4.2.3.1 Alignment Control

Before two floating point numbers can be added or subtracted, they must be aligned with respect to one another so that their radix points meet:

$$\begin{array}{rl} & (145.4_8 \bullet 2^{17}) \times 2^{-17} \\ + & (0.4631_8 \bullet 2^{24}) \times 2^{-24} \end{array}$$

$$= \quad \begin{array}{rl} & 145.40000\,| \\ + & .46314\,|631 \\ \hline & 146.06314\,|631 \end{array}$$

The program has access to the alignment shift limit register (ASL) via a HARDWARE declaration and an ordinary assignment statement. The limit which is stored in the ASL is compared to the actual shift count required to align the numbers (differences of the exponents) after the operation has been completed. If the bound is exceeded, the ASFF (alignment shift flip-flop) is turned on. This flip-flop which is also accessible via a hardware declaration may be tested in an ordinary Boolean test.

4.2.3.2 Normalization Control

Once a non-integer floating point number has been computed, it must be normalized. A register and flip-flop similar to those described above is available for detecting computations in which an excessive normalization shift occurs. These are the NSL (normalization shift limit) and NSFF (normalization shift flip-flop).

4.2.3.3 Floating Point Overflow and Underflow

These floating point faults are detected as interrupt conditions which may be enabled or disabled and for which the programmer may provide contingency actions via the ON-statement group. (See Section 4.11) .

Floating point overflow results when a floating point single or double precision operation produces a result for which the exponent exceeds 63. Floating point underflow results when such an operation produces a result for which the exponent is less than -63. In either case, the signed, normalized and possibly double precision mantissa is left in rB, the out-of-range exponent is left in rA in integer format (sign in bit 0) and the original operands are pushed down into the memory portion of the stack, all these actions taking place before the interrupt action is undertaken.

Example: The following AGC coding could be used to treat a floating point overflow by the rule: halve the exponent and increase a counter by 1; also count up the number of times the out-of-range exponent was odd.

```
ØN FPØ          "ASSUMED SINGLE PRECISION"

DIRECT          Upon entry RPW2 with a zero count field is at TOS

     LFS 4      TOS = 4, RPW2

     OR         RPW2 now has a count field = 4

     FSC 2      Fetch the out-of-range exponent at 2 relative to the CSB.

     STC XPON   Store at XPON; purge from TOS.

     FSC 3      Fetch the mantissa at 3 relative to the CSB.

     STC MANT   Store at MANT; purge from TOS.

END
                                   Divide XPON by 2 and set
.REMQUO (XPØN, 2 = RDR, XPØN)      RDR to the remainder


MANT(S 1//7) = XPON                Insert XPON into exponent field
                                   of MANT


NTXH = NTXH + 1                    Count up number of times exponent
                                   has been halved


NØDD = NØDD + RDR                  Count up number of times exponent
                                   was odd


DIRECT

     FSS MANT   Leave altered result at TOS

END

END
```

The last END causes the generation of the NSX return.

4.3 Instruction Fetch Cycle

Control stream (instruction) words will be fetched one at a time with one-word look-ahead into a wrap-around 64-bit Instruction Register, IR, consisting of IR_0 and IR_1, left and right halves respectively. The syllable which is to be executed is transferred from IR to the Control Stream or Current Syllable register, CS, which is 16 bits wide and holds a one or two-byte (8 bits) instruction syllable. The word address of the next syllable to be executed (if the current syllable doesn't cause a transfer of control) is contained in the 15-bit Next Syllable register, NS. There are in addition, two 3-bit registers: CB which contains the Current Byte and NB which contains the Next Byte. The byte numbers contained in these two registers range from 0 thru 7 and correspond to the numbering of the bytes in the 8-byte IR register.

An additional one-bit flip-flop ILFF contains the length of the current syllable minus 1.

The registers discussed above are illustrated in Figure 4-2 together with bit and byte numbering conventions.

The architectural approach allows instruction syllables to be stored in memory without regard to word boundaries.

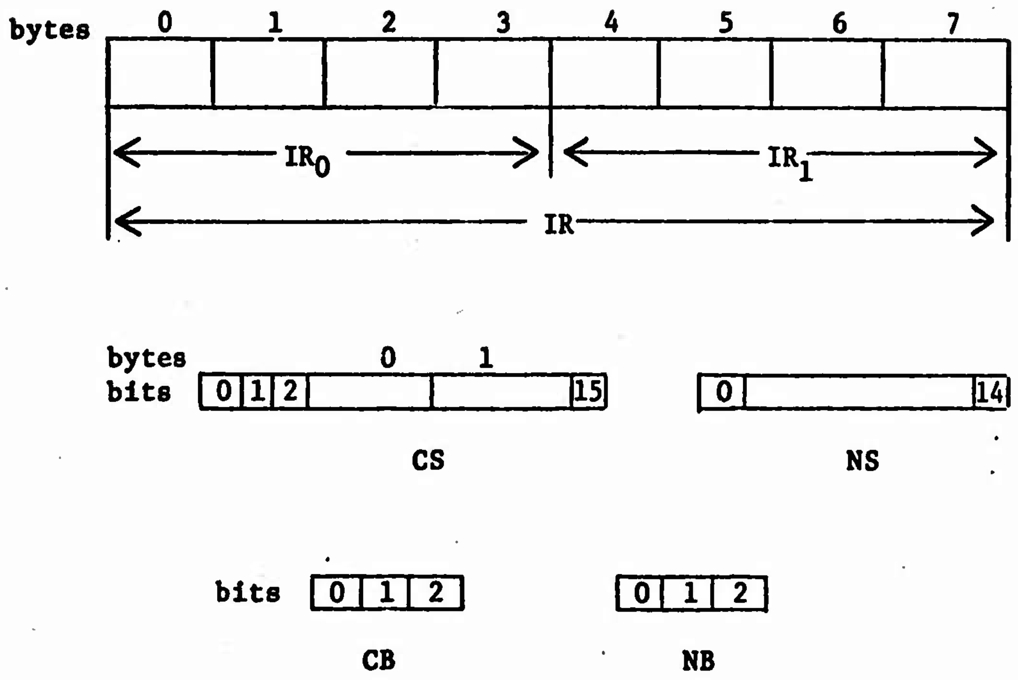


Figure 4-2 : Instruction Fetch Registers

IR = Instruction Register. Holds two words at a time.

CS = Control Stream or Current Syllable register. Holds syllable being executed.

NS = Next Syllable register. Holds word address of next syllable to be executed.

CB = Current Byte number in IR of leading byte in CS.

NB = Next Byte number in IR of leading byte of syllable which will be executed next.

In addition to these registers, we use for convenience of description the following virtual registers in the balance of this report:

PC = Program Counter. This is a 17-bit virtual register giving the word (bits 0-14) and byte (bits 15-16) address of the next syllable to be executed. The first 15 bits are the contents of NS. The last two bits are bits 1 and 2 of NB.

NW = Next Word. This is a 15-bit virtual register giving the address of the next word which will be loaded into either IR_0 or IR_1. NW is really NS+2.

IL = Instruction Load flip-flop. This is a one-bit virtual flip-flop. When it = 0, IR_0 is next to be loaded from memory and when it = 1, IR_1 is next. It is really bit 0 of CB.

In program execution, when a transfer of control occurs, the formula PC ← Address is to be interpreted as, NS ← Address bits 0-14, NB bits 1 and 2 ← Address bits 15 and 16, and NB bit 0 ← 0.

4.3.1 Fetching an Instruction

1. Load Byte 0 of CS with Byte NB of IR, set ILFF = 0.

2. Inspect appropriate bits of CS to determine if a second byte is required for this syllable. If so, load Byte 1 of CS with Byte (NB + 1) Mod 8 of IR, set ILFF = 1.

3. Replace CB register by NB register.

4. Set NB to (CB + ILFF + 1) Mod 8.

5. If CB is < 4 and NB $\geq$ 4, load IR_0 with word at address NW, set NS to NS + 1.

   If CB is $\geq$ 4 and NB < 4, load IR_1 with word at address NW, set NS to NS + 1.

6. Execute syllable in CS.

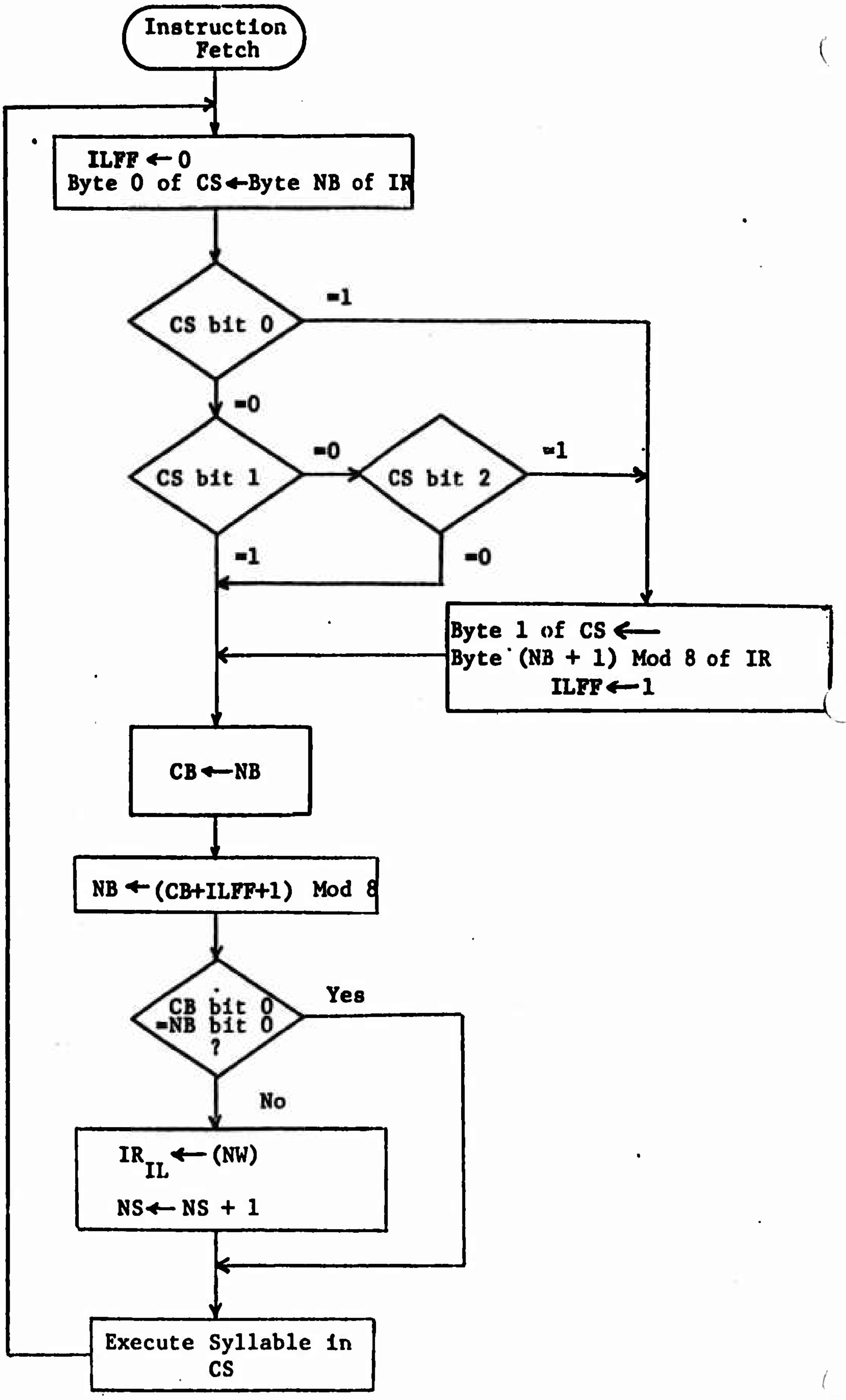


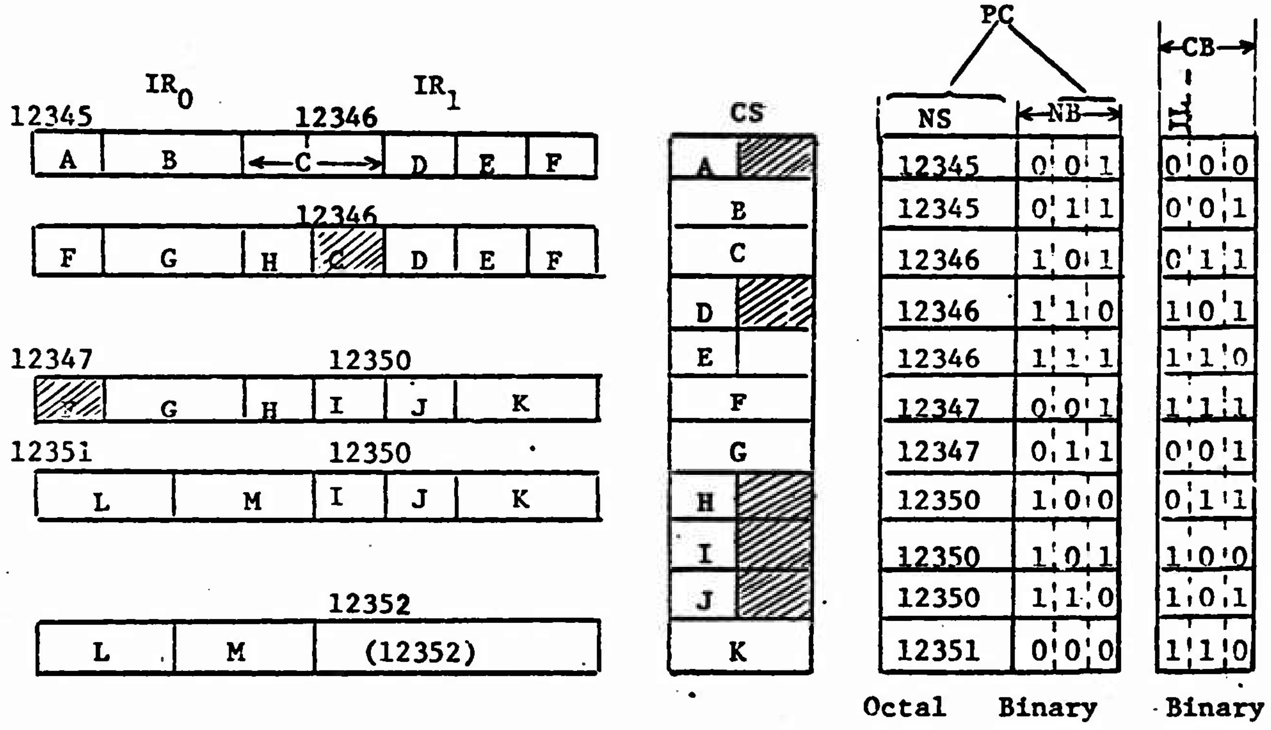
Figure 4-3 : Instruction Fetch Cycle

Example:

A program segment starting at Byte 0 of address 12345 is to be executed. The syllables are labelled with capital letters as follows. Note that two-byte syllables (which may contain a ten-bit address as the last ten bits) can be split over word boundaries:

<table>
<tr><td>12345</td><td>A</td><td colspan="2">B</td><td>C</td></tr>
<tr><td>12346</td><td>C</td><td>D</td><td>E</td><td>F</td></tr>
<tr><td>12347</td><td>F</td><td colspan="2">G</td><td>H</td></tr>
<tr><td>12350</td><td>I</td><td>J</td><td colspan="2">K</td></tr>
<tr><td>12351</td><td colspan="2">L</td><td colspan="2">M</td></tr>
</table>

Figure 4-4: Example of Syllables in Memory



CS	PC: NS (Octal)	PC: NB (Binary)	CB: IL (Binary)	
A ▨	12345	0 0 1	0 0 0	
B	12345	0 1 1	0 0 1	
C	12346	1 0 1	0 1 1	Load IR_0 with (NW) = (NS+2) = (12347)
D ▨	12346	1 1 0	1 0 1	
E	12346	1 1 1	1 1 0	
F	12347	0 0 1	1 1 1	Load IR_1 with (12350)
G	12347	0 1 1	0 0 1	
H ▨	12350	1 0 0	0 1 1	Load IR_0 with (12351)
I ▨	12350	1 0 1	1 0 0	
J ▨	12350	1 1 0	1 0 1	
K	12351	0 0 0	1 1 0	Load IR_1 with (12352)

Figure 4-5 : Instruction Fetch Registers During Execution of Syllables Shown in Figure 4-4

4.4 The Stack Mechanism

The stack mechanism consists of

- two high-speed arithmetic registers to which we attach the temporary names AR_0 and AR_1. These are to be considered 32 bits wide in single precision operations and 64 bits wide in double precision;
- two one-bit flip-flops F_0 and F_1 which correspond to AR_0 and AR_1 respectively. These flip-flops are used to denote whether (=1) or not (=0) the corresponding arithmetic register is empty;
- a one-bit flip-flop c which indicates which of the pairs (AR_0, F_0) or (AR_1, F_1) corresponds to the top of the stack;
- an eleven-bit stack pointer register P which contains the address of the highest address (top of the memory portion)of the stack. P is altered during ordinary stack operations in a manner to be explained below.
- the stack area of scratch pad memory. This is the area pointed at by P and i3 allocated by the compiler/executive system.

The topmost arithmetic register is referred to as rA or the A register and the other is called rB or the B register as follows:

If c = 0, AR_0 is called rA and F_0 is called a,

AR_1 is called rB and F_1 is called b.

If c = 1, AR_1 is called rA and F_1 is called a,

AR_0 is called rB and F_0 is called b.

The operation of the stack will be illustrated by examining closely two instructions:

FSS Y : Fetch a single precision operand from scratch pad memory.

Y is the address of the operand.

Binary Compositions

e.g. + : Add rA to rB. Leave the result in rB and mark rA empty.

This is typical of most operations.

After these illustrations, we shall drop the notation AR_0, AR_1, F_0, F_1 in the balance of this report and use only A, B, a, b.

4.4.1 Fetching an Operand to the Stack (FSS)

In SPL-style notation the rules are

```
IF rA is empty THEN fill it and mark it filled ELSE
IF rB is empty THEN fill it, mark it filled and invert the
                              registers (rA becomes rB becomes rA)
   ELSE empty rB into memory and proceed as though rB had been empty
ENDALL
```

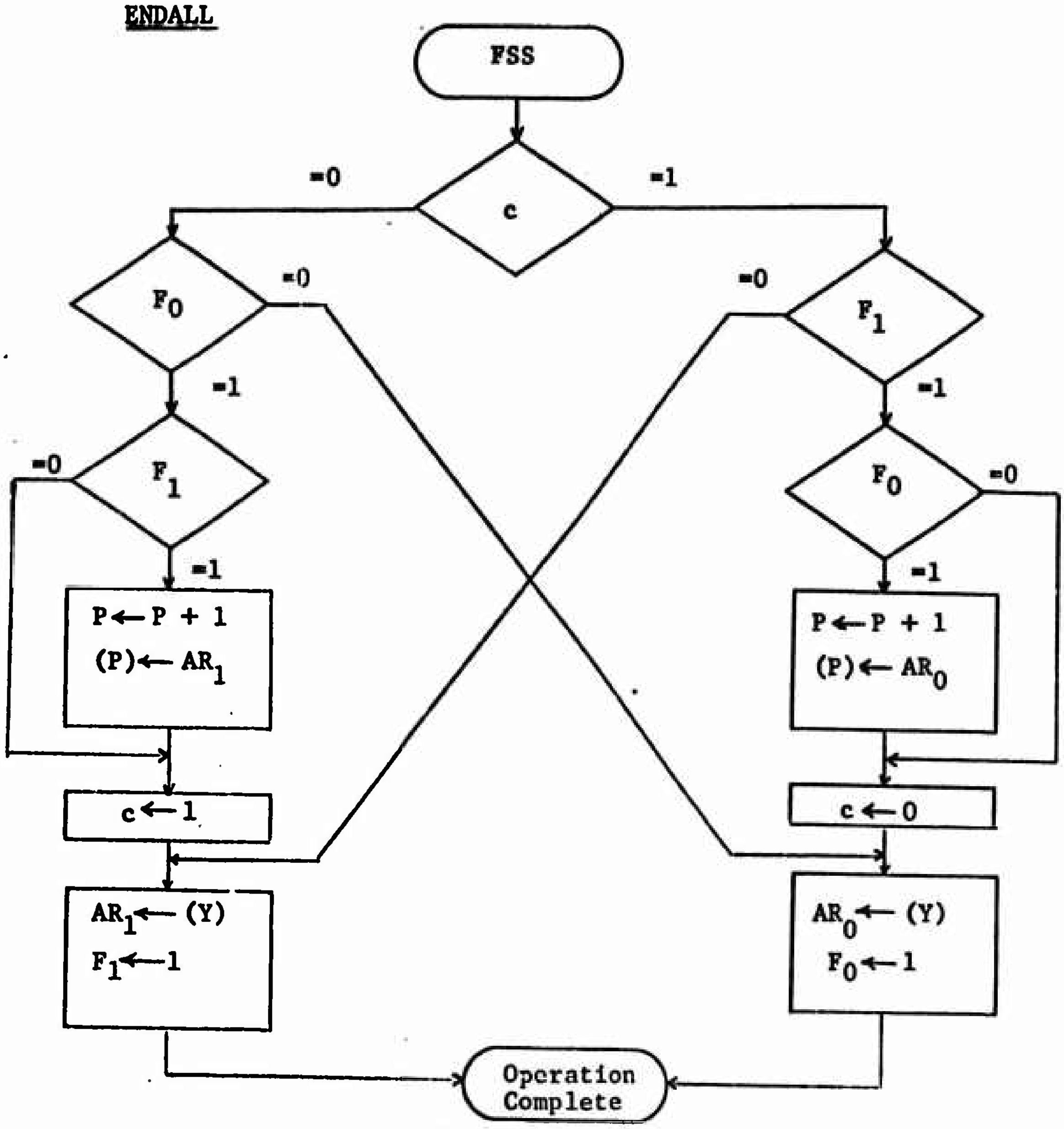


Figure 4-6: Operand Fetch

Thus, starting with an empty stack, the first two fetches fill the registers A and B while a third fetch causes the contents of B to be stored in memory to make room for the new operand. The last fetched (or computed) quantity is always on the top of the stack and is the first entity available for storing. Such stacks are called LIFO's for last in first out.

Example: Fetch X, Y, Z in that order. Assume stack empty, c = 0 and P = 1000.

Register	Event	Initial Condition	After FSS X	After FSS Y	After Fss Z
c	Contains	0	0	1	0
F_0	Is called	a	a	b	a
	Contains	0	1	1	1
F_1	Is called	b	b	a	b
	Contains	0	0	1	1
AR_0	Is called	rA	rA	rB	rA
	Contains	...	X	X	Z
AR_1	Is called	rB	rB	rA	rB
	Contains	...	...	Y	Y
P	Contains	1000	1000	1000	1001
(P)	Contains	...	...	...	X

Notes: 1. (P) is the contents of the address contained in P

2. ... denotes empty or unspecified.

4.4.2 Binary Compositions (+, -, *, ÷, etc.)

All arithmetic and Boolean operations take the form rB op rA → rB mark rA empty:

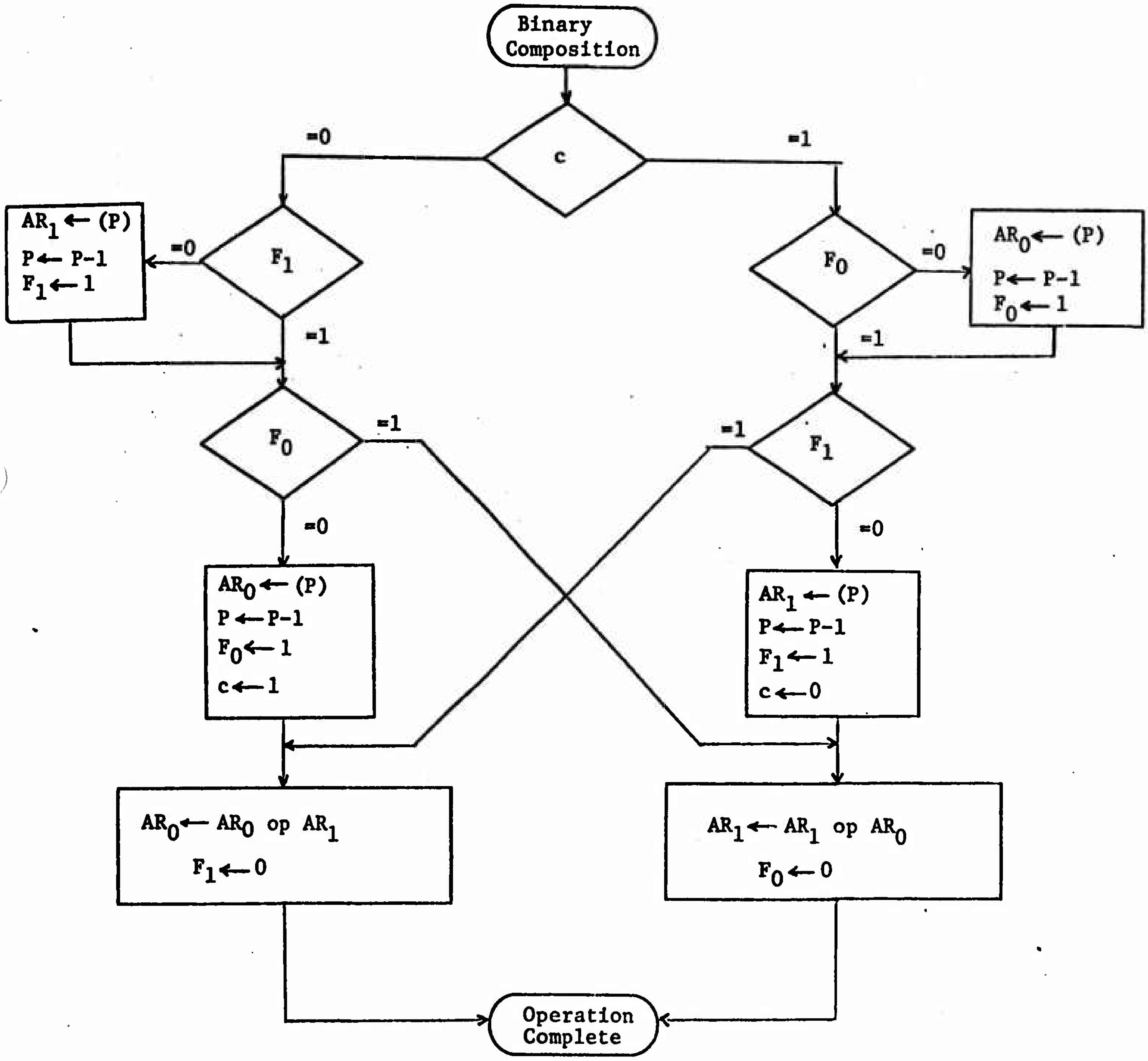


Figure 4-7

To summarize, if we number the elements on a stack as ① for topmost, ② next, etc., then ignoring the c flip-flop we have

Stack Contents

a	b	①	②	③
1	1	rA	rB	(rP)
1	0	rA	(rP)	(rP-1)
0	1	rB	(rP)	(rP-1)
0	0	(rP)	(rP-1)	(rP-2)

The topmost position or its contents will often be referred to below as TOS (top of stack).

4.4.3 Stack Behavior with Double Precision Operands

When a double precision number is fetched to the top of the stack, the most significant half (the word addressed) goes to the left half of rA and the least significant half (at address + 1) goes into the right half of rA. This causes the APFF (rA precision flip-flop) to be turned on. Similarly, when a double precision number is to be stored in the memory portion of the stack, the most significant half is stored first followed by the least. Thus, rP ends up pointing to the least significant half of a double precision number. Then, when a double precision arithmetic operation requires a stack adjustment (i.e., filling of the arithmetic registers) before it can proceed, the least significant half is fetched to the right and the most significant half to the left half of rB, setting the BPFF (rB precision flip-flop) and counting rP down twice.

4.5 Instruction Syllables - Scalar Mode

The AGC processor allows for two modes of operation, the scalar and array modes. Scalar mode is the mode in which ordinary arithmetic, Boolean and logical computations are performed on one or two-word data items including individual elements of arrays. Scalar mode also allows for input, output, calling of subroutine and function subprograms and for operations on partial words. Array mode allows for operations on whole arrays (vectors and matrices) and array cross-sections (rows or columns selected from a matrix). The scope of operations includes element-by-element operations between two arrays or cross-sections (result element = simple arithmetic binary composition of two elements), scalar operations (result element = simple arithmetic binary composition of a scalar and an array element in either order), array multiplication (which may produce a scalar result) and vector cross product. Array mode does not permit input, output, calls of subprograms or partial word operations.

Scalar mode is entered by execution of the ESM (enter scalar mode) syllable and array mode by the EAM (enter array mode) syllable (Table 4-1). These operators are valid in both modes.

Instruction syllables are of length 1 or 2 bytes which may be stored without regard to word boundaries. They are of four distinct types as shown in Table 4-1. Each type is discussed at length in the following sections.

Syllable Type	Length (bytes)	First Two Bits	Purpose
Literal Fetch	1 or 2	00	Bring integers ≤ 8191 into TOS.
Operator	1	01	Perform no-address operations.
Addressing Operator	2	10	Address operations. Address operands directly.
Descriptor Call	2	11	Address operations. Address arrays, subroutines, etc., indirectly via descriptor words.

Table 4-1 : Control Stream Syllable Types -
Scalar and Array Mode

4.5.1 Literal Fetch Syllable (LFS)

An LFS is used to load a positive integral value to the top of the stack. This is useful in two regards

- first, to fetch operands I: $|I| \leq 8191$ used in floating point and integer arithmetic computations.
- second, to fetch twelve-bit address displacements to the top of the stack for conditional and unconditional jumps.

The use of the LFS results in considerable program compression in that an LFS fetches a literal in one or two bytes whereas six bytes would otherwise be required: 4 bytes to store the literal as a whole word item and 2 bytes to fetch it with an addressing operator (see below).

An LFS can be one or two bytes in length as follows

| 0 0 0 | X X X X X | for literals ≤ 31

| 0 0 1 | X X X X X X X X X X X X X | for literals ≤ 8191

<u>Examples</u>:

1. FØR I = 0 TØ 29 BY 1

As will be seen later, this SPL language form requires the values 1, 29, 0 (in that order) to be placed on top of stack. A part of the appropriate coding is, therefore,

$$\left.\begin{array}{ll}\text{LFS} & 1\\ \text{LFS} & 29\\ \text{LFS} & 0\end{array}\right\}\ \text{length 3 bytes}$$

2. GØTØ STLAB

Suppose that STLAB turns out to be the third byte (byte 2) of word 903 relative to the start of the program segment containing the GØTØ.

The appropriate coding with commentary is

length 3 bytes	LFS	7036_8	This is 4*903 + 2
	JMP		Last 17 bits of TOS → PC

4.5.2 Operator Syllable

An operator syllable is always one byte in length and is used to perform all arithmetic, Boolean, logical, manipulative, storage and control functions for which (because the operands generally are at the top of the stack) no address is required or for which (as in the immediately preceding example) an address is already present at the top of the stack.

The format of an operator syllable is

0 1	f f f f f f

where f is specific to the operation to be performed. The following tables show the scalar mode operations classified by general type. No assignment of operator has been made except for ESM (f = 00_8) and EAM (f = 77_8).

Symbol or Mnemonic	Function	Single	Double	Comments
+	rB + rA → rB	X	X	
-	rB - rA → rB	X	X	
÷ R	rB ÷ rA → rB	X	X	Real
÷ i	[rB ÷ rA] → rB rB Mod rA → rA	X		Operands are integers or are made so.
*	rB * rA → rB	X	X	
CHS	-rA → rA			
ABS	|rA| → rA			
RND	Round rA at word boundary			sign is retained
TRC	Singularize double to single precision by truncation			
INT	Replace rA by its integer part			
FRAC	Replace rA by its fractional part			
MIN	Min {rA, rB} → rB			
MAX	Max {rA, rB} → rB			
SGN	If rA $\left\|\begin{matrix} < 0 \\ \geq - \end{matrix}\right\|, \left\|\begin{matrix} - \\ + \end{matrix}\right\|$ 1 → rA			Signum function

Table 4-2: Scalar Operations
Arithmetic Functions: All
operations are floating point
unless otherwise noted.

Symbol or Mnemonic	Function	Comments
AND	rB ∩ rA → rB	Bit by bit for 32 bits
ØR	rB ∪ rA → rB	
XØR	rB ∇ rA → rB	
EQUIV	rB Δ rA → rB	
CMP	C_1(rA) → rA	Logical Not
NØT	0 → rA (0//31) and Complement last bit:	Boolean Not.
XCH	rA ⇄ rB	Exchange operator.
DUP		Duplicate top of stack.
NIX		Delete word in TOS.
EXT	Partial word Operations	Extract a partial word.
SXT		Extract a partial word and sign.
INS		Insert a partial word.
SNS		Insert a partial word and sign.

Table 4-3: Scalar Operations
Boolean, Logical, Manipulative Functions

Symbol or Mnemonic	Function	Comments
GR	≡ REL	If rB REL rA is true, a Boolean true (integer 1) → rB If rB REL rA is false, a Boolean false (integer 0) → rB. In either case both comparands are deleted from the stack.
GQ		
NQ		
EQ		
LQ		
LS		
JMP	Jump to address in rA	unconditionally
JØT		if rB contains true false
JØF		
NSX		Normal Subroutine Exit
ASX		Abnormal Subroutine Exit
NDX		Transfer of control at end of FOR-loop (conditional)
INX		Transfer of control at end of FOR-loop (unconditional)

Table 4-4: Scalar Operations
Comparison & Branching Functions

Symbol or Mnemonic	Function	Comment
FIC	TOS ← ICR	Fetch Interrupt Condition Register to TOS
SIC	TOS → ICR	Store TOS in Interrupt Condition Register
FIM	TOS ← IMR	Fetch Interrupt Mask Register to TOS
SIM	TOS → IMR	Store TOS in Interrupt Mask Register

Table 4-5 : Scalar Operations
Interrupt Processing

Symbol or Mnemonic	Function	Comments	
←C	rB→Address in rA	Single Precision	Both rA, rB are cleared from stack.
⇐C	rB→Address in rA	Double Precision	
←K	rB→Address in rA	Single Precision	Only rA is cleared from stack. rB is kept.
⇐K	rB→Address in rA	Double precision	
FCV	Current value of innermost FOR-loop control variable (contents of CVR register)→TOS		
SCV	TOS→CVR		
PCV	CVR→Address in the ACVR register		
XCR	Index create. Set up new values in the FOR-loop control registers (ACVR, CVR, FVR, IVR) after preserving their current contents and the loop return point.		
EAM	Enter Array Mode		
ESM	Enter Scalar Mode		
PRR	Program Record (see section on circumvention).		
PCR	Program Recover Return (see section on circumvention).		
NOP	Pick up next syllable.		

Table 4-6: Scalar Operations
Storing, Indexing & Miscellaneous Functions

Example:

In the following example, the mnemonics for the required addressing operators are written out.

Compile Y = X * (Y - 2/W)

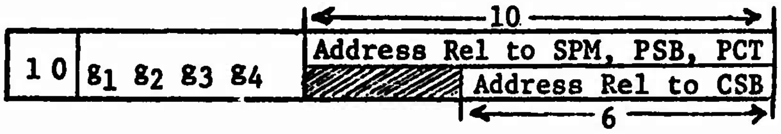
Length	Mnemonic	Address	Comments
2	LFS	A(Y)	Literal giving address of Y relative to the scratch pad memory
2	Fetch	X	
2	Fetch	Y	
1	LFS	2	
2	Fetch	W	
1	÷R		2/W
1	-		Y - 2/W
1	*		X * (Y - 2W)
1	XCH		Interchange result with A(Y)
1	←C		Store Clear Single
14 bytes			

When we have discussed the addressing operators, we shall repeat this example reducing its length by 2 bytes by using the storage addressing operators. The left arrow operator used here (see Table 4-6) is intended for subroutines wherein a result is known by address: Compute result, fetch address argument, left arrow.

4.5.3 Addressing Operator Syllable

An addressing operator is always two bytes long with the last six or ten bits constituting an address relative to one of the four base address registers SPM (Scratch Pad Memory), PSB (Program Segment Base), PCT (Program Control Table) and CSB (Calling Sequence Base). These matters are discussed at greater length in sections on memory structure, addressing. program segmentation and subroutine entry.

The format of an addressing operator is



where g is specific to the function to be performed.

The function of an addressing operator is to fetch to the TOS a one or two-word data item where the address refers directly to the item (if one word) or to its first word (if two words).

The g-bits are encoded as follows:

g_1 = 0 for fetching

= 1 for storing

g_4 = 0 for single precision

= 1 for double precision

g_2 g_3 when g_1 = 0 (fetching operations)

= 00 fetch relative to SPM

= 01 fetch relative to PSB

= 10 fetch relative to PCT

= 11 fetch relative to CSB

g_2 g_3 when g_1 = 1 (storing operations)

g_2 = 0 store by truncating

= 1 store after rounding the TOS continuation register into the TOS most significant half.

g_3 = 0 clear the operand from TOS after storing it.

= 1 keep the operand on TOS after storing it.

This scheme results ir two meaningless combinations 1101 (store double round clear) and 1111 (store double round keep). Accordingly, the first of these is used to store a single precision zero and the second to store a double precision zero at the given address. All storage addresses are relative to SPM. The scalar mode addressing operators are summed up in Table 4-7.

Mnemonic	g-Bits	Function	
FSS	0 0 0 0	Fetch Single from SPM	
FDS	0 0 0 1	Fetch Double from SPM	
FSP	0 0 1 0	Fetch Single from PSB	
FDP	0 0 1 1	Fetch Double from PSB	
FST	0 1 0 0	Fetch Single from PCT	
FDT	0 1 0 1	Fetch Double from PCT	
FSC	0 1 1 0	Fetch Single from CSB	
FDC	0 1 1 1	Fetch Double from CSB	
STC	1 0 0 0	Store (Single) Truncate Clear	in SPM
SDC	1 0 0 1	Store Double Clear	
STK	1 0 1 0	Store (Single) Truncate Keep	
SDK	1 0 1 1	Store Double Keep	
SRC	1 1 0 0	Store (Single) Round Clear	
SSZ	1 1 0 1	Store Single Zero	
SRK	1 1 1 0	Store (Single) Round Keep	
SDZ	1 1 1 1	Store Double Zero	

Table 4-7 : Scalar Addressing Operators
- Those with $0 \leq g \leq 7$ are also
Recognized in Array Mode

Examples:

1. Compile Y = X * (Y - 2/W)

Length	Mnemonic	Address	Comments
2	FSP	X	X is local to program segment.
2	FSS	Y	Y is in SPM
1	LFS	2	
2	FST	W	W is global to all programs
1	÷R		2/W
1	-		Y - 2/W
1	*		X * (Y - 2/W)
2	SRC	Y	Round & Store Single Precision Result & clear from TOS.
12			

2. Compile PHI = THETA - OMEGA = ALFA * PI

Length	Mnemonic	Address	Comments
2	FSS	ALFA	
2	FST	PI	Global Constant
1	*		
2	SRK	OMEGA	Store & Retain for further use.
1	CHS		- OMEGA
2	FSS	THETA	
1	+		
2	STC	PHI	PHI declared with TRUNCATE attribute.
13			

4.5.4 Descriptor Call Syllable

A descriptor call syllable is likewise two bytes long with the same address structure as an addressing operator except that the address pertains only to the PCT (ten bits) or the CSB (six bits). The entity referred to by the address is not the data itself but a word which describes the data. These words, called descriptors, are of several different types of which two will be described later in this section by their use in examples.

The execution of a descriptor call involves the following steps:

1. Recognize descriptor call.

2. Fetch the descriptor into the Descriptor Decoding Register (DDR).

3. Depending on the particular descriptor call, distribute the partial fields of DDR to other control registers for use during execution.

4. Execute the operation implied in the descriptor call.

The DDR is a description of a function which may be assumed by a part of the microprogramming logic.

The format of a scalar mode descriptor call is

<table>
<tr><td>0 1</td><td>2</td><td>3 4 5</td><td colspan="2">6 7 8 9 10 11 12 13 14 15</td></tr>
<tr><td rowspan="2">1 1</td><td>0</td><td rowspan="2">h h h</td><td colspan="2">Address Relative to PCT</td></tr>
<tr><td>1</td><td>▨</td><td>Addr Rel to CSB</td></tr>
</table>

where h is specific to the function to be performed, and bit 2 determines whether the address is relative to the PCT or the CSB for $0 \leq h \leq 5$.

Table 4.8 presents the scalar mode descriptor calls.

Mnemonic	h-Bits	Function
ELF	0 1 0 0 0	Fetch an element from a one or two-dimensional array. Indices are at TOS and dimensions, base address & storage mode of array are contained in an array descriptor. One may not use two indices if either dimension is 1.
ESC	0 1 0 0 1	Store an element in a one or two-dimensional array. Clear TOS of indices and element.
ESK	0 1 0 1 0	Same as ESC, but data element is saved.
IN	0 1 0 1 1	Initiate input according to File Descriptor.
ØUT	0 1 1 0 0	Initiate output according to File Descriptor.
ESP	0 1 1 0 1	Enter Subroutine Program.
JOS	0 1 1 0	Jump to new Program Segment utilizing Program Descriptor in PCT.
DCI	1 1 1 0	Execute the descriptor call stored relative to CSB.
ADE	0 1 1 1 1	Use the element indices at TOS to create and leave at TOS the absolute address of the element.

Table 4-8: Scalar Mode Descriptor Calls

Examples:

```
1.  DECLARE (10, 15) FLOATING, A, B, C
           .
           .
           .
    FØR J = 1 BY 1 TO 14
    FØR I = 1 BY 1 TO 9
    A(I, J) = B (I-1, J) + C(I, J-1) ENDALL
```

We defer consideration of the loop set-up and testing syllables until indexing operations have been discussed.

Length	Mnemonic	Address	Comments
1	FCV		I→TOS
1	LFS	1	1→TOS
1	-		I - 1→TOS
2	FSS	J	TOS = J, I - 1
2	ELF	D(B)	B(I-1, J)→TOS
1	FCV		I→TOS
2	FSS	J	J→TOS
1	LFS	1	J - 1→TOS
1	-		TOS = J - 1, I
2	ELF	D(C)	TOS = C(I, J-1), B(I-1, J)
1	+		B(I-1, J) + C(I, J-1)→TOS
1	FCV		I→TOS
2	FSS	J	TOS = J, I, B(I-1, J) + C(I,J-1)
2	ESC	D(A)	B(I-1, J) + C(I, J-1)→A(I, J)
20			

In this example, D(A), D(B), D(C) are the addresses of the array descriptors of A, B, C respectively. Without implying their consecutive storage these have the following format:

				4	5	5	15
D(A)	0	1	0	▨	9	14	Address of A(0, 0)
D(B)	0	1	0	▨	9	14	Address of B(0, 0)
D(C)	0	1	0	▨	9	14	Address of C(0, 0)

← 32 bits →

Bit 2 gives the storage mode (0 = single precision, 1 = double precision).

Example:

2. Call a subroutine

Y = .SIN (W + Z)

Length	Mnemonic	Address	Comments
2	FSS	W	W→TOS
2	FSS	Z	Z→TOS
1	+		Leave value of argument at TOS
2	ESP	D(.SIN)	Compute sine
2	SRC	Y	Store, round, clear

The subprogram call descriptor D(.SIN) has the following format:

			9	6	15
D(.SIN)	1	1	▨	1	Address of first word of subroutine

↑ # of words in argument list

4.6 Partial Word Operations

4.6.1 Language Considerations

Syntax: Partial word operations are permitted in SPL/MK III through the use of the bit modifier which is written as

- a parenthesized pair of fields which can be
  - literals
  - variables
  - arithmetic expressions

  with the fields separated by a pair of slashes.

- an alternative form in which the literal letter "S" appears between the opening parenthesis and the first field described above.

The first field gives the starting bit number, SB:

$0 \leq SB \leq W - 1$ where W = word length in bits and bits are numbered 0 to W - 1 left to right.

The second field imparts the length of the partial word field, LF:

$1 \leq LF \leq W$

Note also that the following condition must obtain:

$SB + LF \leq W$

The word length W for the AGC is 32 and the balance of these discussions assume this value.

Examples:

1. D = 6 * A (S 0//16)
2. D (8//8) = BYTE 1

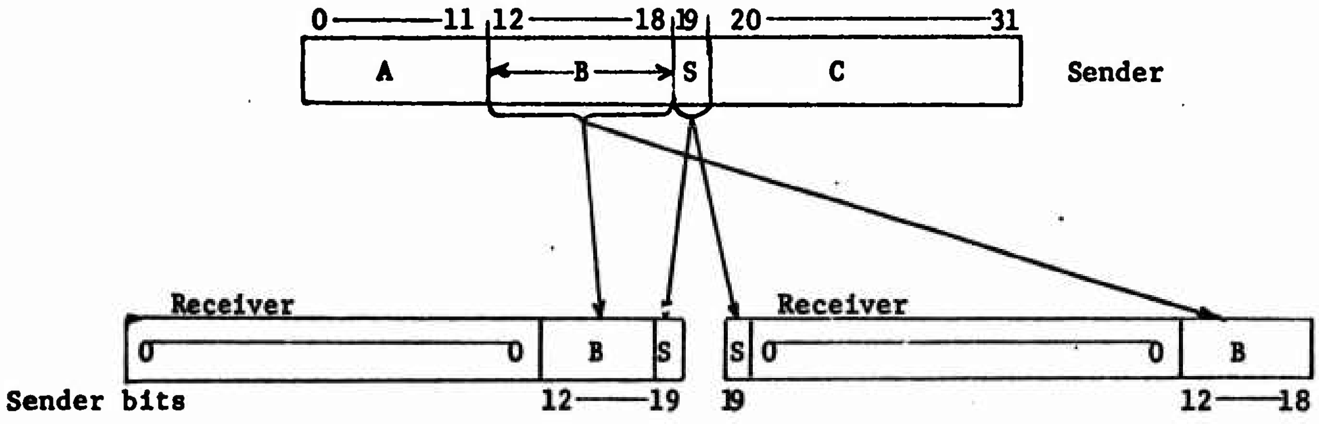
Semantics: Extraction: A bit modifier appearing anywhere in an arithmetic or logical formula is regarded as a postfixed unary operator calling for extraction. As a unary operator, the usual rules for the scope of its effect apply. On the AGC this means that it is applied to the quantity at the TOS without regard to how the latter got there. In practical terms, this means that partial word fields can be extracted from parenthesized expressions and used as operands in further computation:

A = (P + Q * R)(S 1//7) + 64

This expressions calls for the signed floating point exponent of the quantity P + Q * R to be added to 64. As a unary operator, extraction has higher precedence than any other operator except the prefixed unary operators +, -.

4.6.1.1 Extraction

Extraction in SPL/MK III means field extraction: The fields adventitious to the field specified by the bit modifier are zeroed out and then the specified field is right justified and left zero filled. The alternative form of the bit modifier with the letter "S" calls for signed field extraction which is the same as field extraction except that one of the bits of the extracted field is put into the sign position of the result. On the AGC, the last bit of the sender field is stored as the sign (bit 0) of the rece:ver:

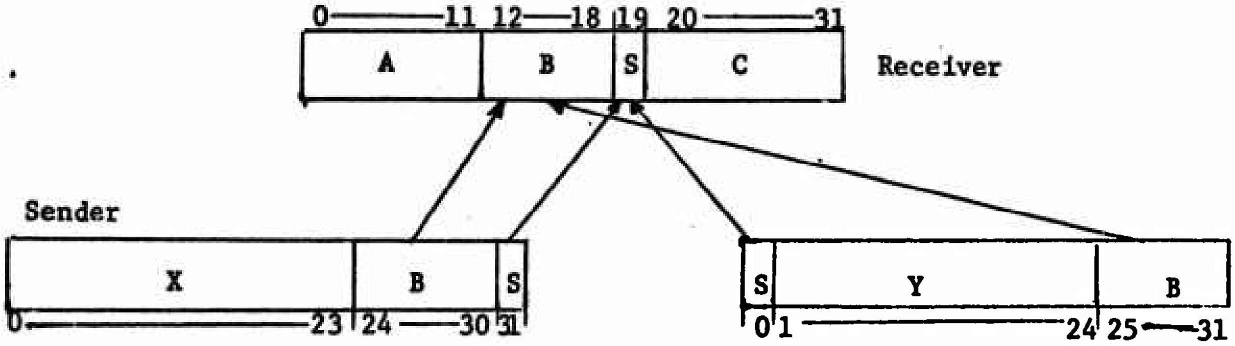


Field Extract (12//8) Signed Field Extract (S 12//8)

A further form of extraction which removes the adventitious fields but leaves the extracted field in place is termed logical extraction. It may be achieved under certain circumstances by the logical LAND operation.

4.6.1.2 Insertion:

A bit modifier appearing on the left side of an assignment statement or on either side of an exchange statement is regarded as a modified storage instruction which calls for the (field) insertion of the sender field (last LF bits of the word at the TOS) into field (SB//LF) of the receiver word (the simple or subscripted variable appearing before the bit modifier. Signed field insertion is the same as field insertion except that the sign bit of the sender is combined with the last LF - 1 bits of the sender to make up the field to be inserted. The AGC picture is:



field insert (12//8)

signed field insert (S 12//8)

Note that extract leaves a result on TOS whereas insert causes a result to be stored in memory.

A further form of insertion which inserts the correspondingly positioned field of the sender into the receiver is termed logical insertion. It may be accomplished under certain circumstances by the logical operations LAND, LOR and LNOT.

4.6.2 Compilation Considerations

The methods chosen for implementing extraction and insertion on the AGC are motivated by the need for an easily implemented left-to-right compiler scan of the bit modifier. Thus, assuming for both extraction and insertion that the sender is at the TOS, the basic coding pattern adopted is

- get SB to TOS
- get LF to TOS
- issue extract or insert operator which uses shifting at the microprogram level.

More specifically, the steps for extraction are:

E.1 Set a flag on if "S" appears after "(", otherwise set flag off.

E.2 Drive out code to evaluate SB.

E.3 Drive out code to evaluate LF.

Steps E.2 and E.3 can therefore be

LFS for a literal

FS - (or FCV) for a variable or constant

or a sequence of fetches and arithmetic operators to evaluate a formula.

The net result is that the stack contains at this point:

rA LF

rB SB

(P) Sender

E.4 Issue an EXT (extract) if Step 1 flag is off and an SXT (signed extract) if the flag is on.

The corresponding steps for insertion are:

I.1 Fetch the value of the receiver to the TOS with an FSS.

I.2-I.4 Same as steps E.1 thru E.3.

The net result is that the stack contains at this point:

rA	LF
rB	SB
(P)	Receiver
(P-1)	Sender

I.5 Issue an INS (insert) if step I.2 flag is off and an SNS (signed insert) if the flag is on.

I.6 Issue an STC to the address of the receiver.

4.6.3 Architecture Considerations

The two methods available for extracting and inserting are masking and shifting. Masking is available through the logical SPL operations LAND, LOR, LNOT and is relevant only under the following conditions:

- a logical extract or insert is desired;
- the parameters SB and LF are both integer literals.

Under these circumstances, the following SPL coding could be used

```
Extract:  REC = SEND LAND MASK
Insert:   REC = SEND LAND MASK LOR REC LAND LNOT MASK
```

LNOT is Ones Complement (Unary operator).

MASK is a constant with ones at the desired field and zeros elsewhere.

When a field extract is wanted, the masking operation must be followed by a shift so that shifting might as well have been used from the start; when either SB or LF is not a literal, a dynamic mask has to be created, presumably by shifting. Thus, there seems to be no way to avoid shifting operations in connection with partial word operations. CIRAD has been unable, however, to identify any other situation in the guidance and navigation computational environment which (given floating point operations) absolutely requires shift instructions. Shift operations are available, therefore, only at the microprogram level for the implementation of the operators shown in Tables 4-2 thru 4-7, including the extract and insert operators which will be discussed below.

4.6.4 Microprogramming Considerations

The following steps suggest one possible way to implement extraction and insertion via shifting operations. (Mu (μ) stands for micro.)

4.6.4.1 Extraction: (EXT or SXT)

μE.1 Integerize rA; rA-1 → rL.

The subtraction allows LF ($1 \leq LF \leq 32$) which may have six bits to be stored in a five-bit register.

μE.2 Adjust Stack; Integerize rA; rA → rS

The integerization of LF and SB allows them to be computed by general arithmetic computations. The integerization is by truncation not rounding. rL and rS are five-bit registers coincident with a pair of the M and N registers of array mode.

μE.3 Adjust Stack. Sender is now in rA.

μE.4 Extract by the following shift operations:
(See Figure 4.8)

μE.4.1 Set shift count from rS. Left shift rA losing the leading SB bits. If SB=0, this is a null operation; this remark applies to the rest of the shifts described below.

μE.4.2 Set shift count from the ones complement of rL (=32-LF). Right shift rA leaving desired field right-justified, left-zero-filled.

μE.5 SXT only: Right circulate rA one place putting the last bit of the field in the sign position.

Note that the total shift count is $32 - LF + SB$. In view of the constraints $0 \leq SB \leq 31$, $1 \leq LF \leq 32$ and $SB + LF \leq 32$, the maximum count of the number of ones bits in any extract is 10 ($SB = 31$, $LF = 1$). This means that the shifting steps in μE.4 can be accomplished in a maximum of 10 clock times.

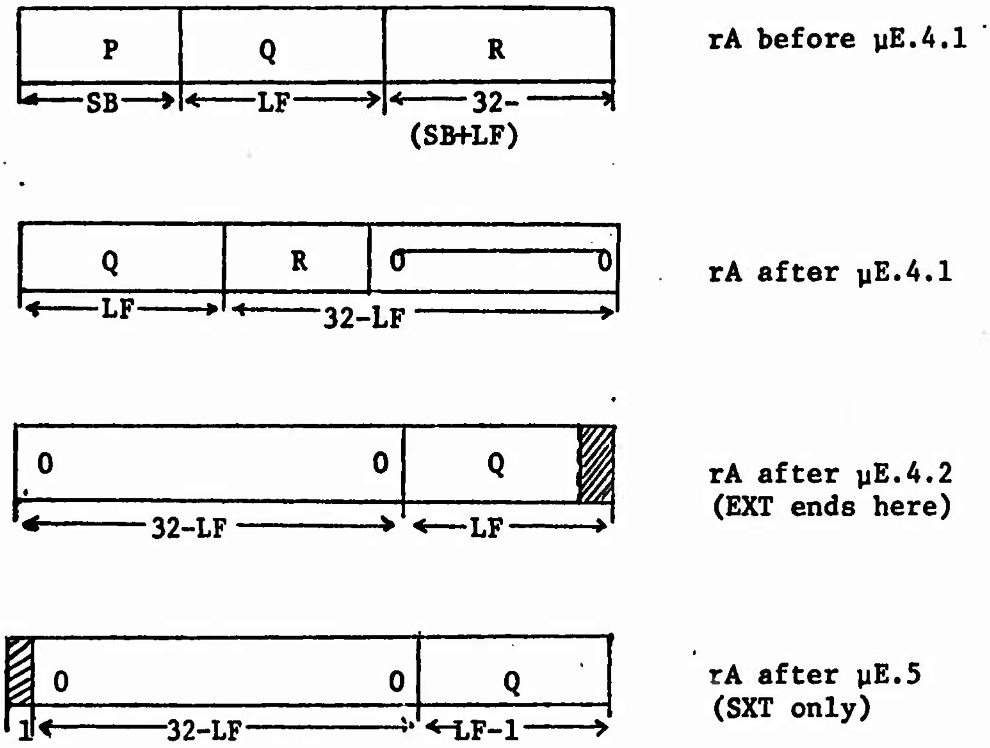


Figure 4-8: EXT and SXT Shifting Steps

4.6.4.2 Insertion: (INS or SNS)

μI.1 - μI.2 Same as μE.1 - μE.2.

μI.3 Adjust Stack. Stack is: rA = Receiver

rB = Sender

μI.4 Considering rA as a double length register with halves rA_L and rA_R.

$$rA_L \rightarrow rA_R$$

$$rB \rightarrow rA_L$$

$$0 \rightarrow b$$

i.e.

$$rA = \boxed{\text{Sender}}\boxed{\text{Receiver}}$$

μI.5 SNS only: Left circulate rA_L 1 place to put sign bit of sender at end of field.

μI.6 Insert by the following shift operations. (See Figure 4-9)

μI.6.1 Set shift count from rS + rL + 1. Left circulate rA_R.

μI.6.2 Set shift count from rL + 1. Double right shift rA_R and rA_R together.

μI.6.3 Set shift count from rS. Right circulate rA_R.

μI.7 $rA_R \rightarrow rA_L$

Note that the total shift count is 2*(SB + LF) for which the maximum ones bit count is 16 (SB = 15, LF = 15). Thus steps μI.4 - μI.7 can be accomplished in a maximum of 20 clock times (21 for SNS).

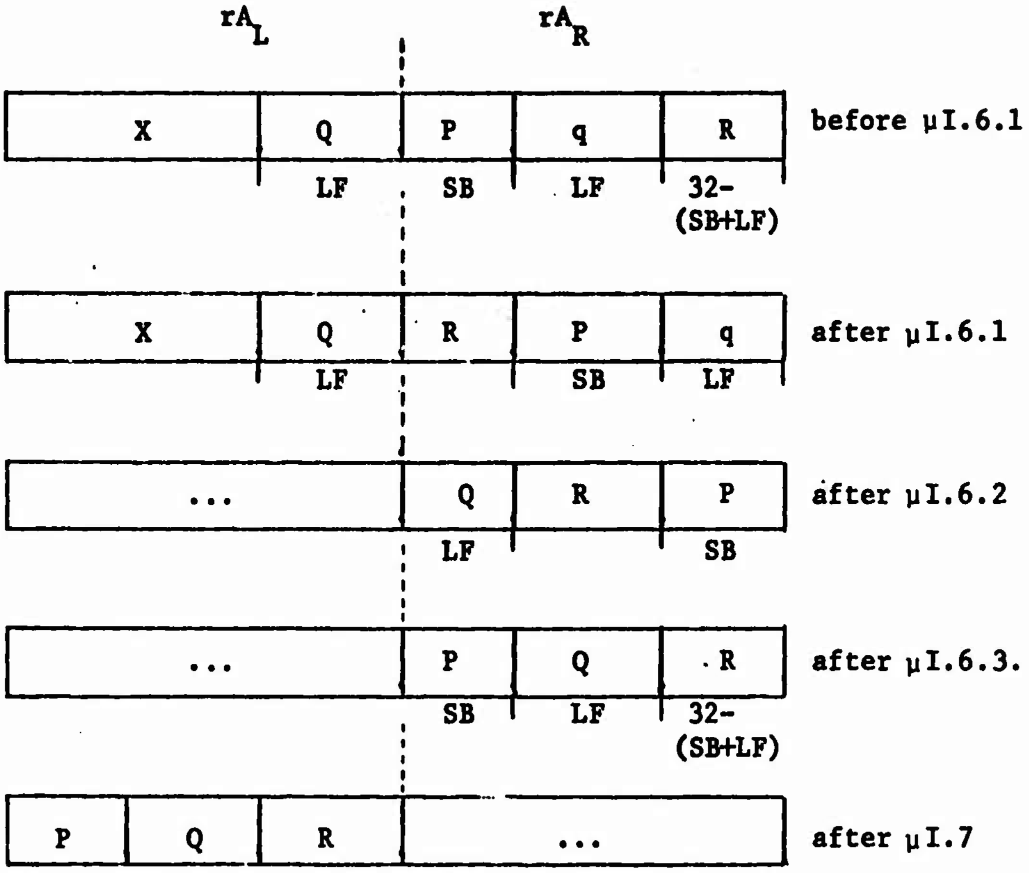


Figure 4-9: INS and SNS Shifting Steps

Examples:

1. Extraction: D = 6 * A(S 0//16)

Length	Mnemonic	Address	Comment
1	LFS	6	Multiplier
2	FSS	A	Sender → TOS
1	LFS	0	Unary Extract operator on A
1	LFS	16	
1	SXT		
1	*		RHS of equation is complete
2	SRC	D	
9			

2. Insertion: D(I + 8//8) = BYTE1

Length	Mnemonic	Address	Comment
2	FSS	BYTE1	Sender → TOS
2	FSS	D	Receiver → TOS
1	FCV		I → TOS
1	LFS	8	I + 8
1	+		
1	LFS	8	TOS = 8, I + 8, Receiver, Sender
1	INS		
2	STC	D	
11			

4.7 Indexing

There are four major uses for index registers in computers of traditional architecture. These are

1. To serve as base addresses for program relocation and for major sections of constant and variable storage.

On the AGC, there is no need for index registers for program relocation as this is not a desirable feature in the guidance computing environment. A group of special base address registers has been provided to allow data to be addressed with six or ten bit addresses. These are the PCT, PSB, CSB and SPM base registers which are discussed elsewhere.

2. For accessing elements of an array.

This need is met on the AGC by loading the subscripts into the arithmetic stack and invoking one of the descriptor calls ELF, ESC or ESK. The descriptor call combines the subscripts in the stack with the base address, dimensions and precision information in the descriptor to create the element address and then fetch the contents thereof to TOS, replacing the subscripts in the stack. In addition, array mode descriptor calls and operations provide for array computations that obviate the need for handling individual elements.

3. For control purposes such as an n-way switch.

The SPL n-way switch takes the form of an indexed GOTO on a list of statement labels which is explicitly declared as a SWITCH or implicitly declared in the GOTO statement itself. The AGC hardware/software solution to this problem is to store the switch list as a vector, creating a descriptor for it. The ELF descriptor call can then be used to fetch the appropriate address to the TOS where it can be operated on by a JMP, JOF or JOT operator. The details are provided elsewhere.

4. For iterative-loop control.

This is the only general indexing capability not otherwise provided for. It is not a pure need, for quite often one needs to use the loop control variable as an arithmetic operand or as an array subscript for an operation which cannot be handled by the array mode operations.

The SPL language need for iteration control is met in two general ways. The first takes the form

$$\left\{\begin{array}{l}\text{WHILE}\\ \text{UNTIL}\end{array}\right\} \quad \text{<Boolean formula>}$$

```
Statement 1
    .
    .
    .
Statement n
```

$$\left\{\begin{array}{l}\text{ENDALL}\\ \text{END}\end{array}\right\}$$

In this form of iteration control it is assumed that the variables entering into the Boolean formula are initialized before the loop is entered and are altered in some manner in the body of the loop so that the termination condition will eventually be fulfilled.

This form of iteration control (Boolean loop control) is easily handled at the compilation level simply by treating the loop as though it had been coded:

```
          LRP.    Statement 1
                      .
                      .
                      .
                  Statement n
     {UNTIL}      IF  {NOT  }<Boolean formula> GOTO LRP
     {WHILE}          {empty}
```

In this form, it can be handled by the comparison and branching facilities already present in the scalar mode operations.

The other major form of iteration control is provided by the concise loop statement which takes the forms

```
          FØR LCV = CV                  (a)
    or    FØR LCV = CV BY IV            (b)
    or    FØR LCV = CV TØ FV            (c)
    or    FØR LCV = CV TØ FV BY IV      (d)
    or    FØR LCV = CV BY IV TØ FV      (e)
    or    FØR LCV = CV, IV, FV          (f)
```

Where the following shorthand notation has been used:

LCV: Loop Control Variable. This is an integer item (non-subscripted).

CV: Commencing Value

or Current Value

IV: Increment Value

FV: Final Value

May be literals, variables or formulas

Forms (d), (e) and (f) are complete and require no further comment except that the signs of IV and FV-CV must agree. Form (c) is treated as though one had appended "BY 1". Form (a) is treated as though one had written "FOR LCV = CV TO CV BY 1", i.e., the loop will be executed exactly once.

When form (b) is used, it must be assumed that the programmer has included his own Boolean test for termination in the body of the loop. For purposes of starting the compilation of such a loop, it is treated as though one had written "FOR LCV = CV TO CV BY IV" and thus treat it like all of the other cases. The loop will, however, be terminated by the equivalent of CV = CV + IV followed by an unconditional transfer to the start of the loop, whereas the other cases follow this incrementation with the equivalent of the test

```
IF IV GQ 0 AND CV LQ FV ØR
   IV LS 0 AND CV GQ FV    GØTØ LRP
```

where LRP is the virtual label of the first statement of the loop (l[illegible] return point).

The following table shows the cost of actually expanding the incrementation and testing steps into AGC code based on the explicit SPL statements shown in the previous paragraph .

	CASES	
Item	a c-f	b
Number of Bytes	30	10
Number of Syllables	22	6
Number of Memory Accesses	7	4

This is reduced in the architecture to 1 byte, 1 syllable and a lesser number of memory accesses which are dependent on implementation (hardware register vs dedicated memory location) by the introduction of the four loop control registers:

# bits	Name	Function
14	CVR	holds CV
14	FVR	holds FV
8	IVR	holds IV
10	ACVR	holds the address of the LCV

This imposes the following constraints on the size of CV, FV and IV:

$0 \leq CV \leq 16383$ (14 bits)

$0 \leq FV \leq 16383$ (14 bits)

$-127 \leq IV \leq 127$ (7 bits and sign)

4.7.1 Setting Up a FOR Loop

The first thing that is required is to put the address of the LCV on the TOS. Since this is a ten bit number (relative to SPM) this can be done with an LFS, or an FSC if the LCV is an output parameter of a subroutine.

Then the quantitites IV, FV and CV are loaded to TOS in that order. This is done as follows:

LFS if literal ≤ 8191

FS- if variable, constant, or literal > 8191

FCV if LCV of next outer FOR loop

or a sequence of syllables to evaluate a formula
and leave result on TOS

Finally, an XCR syllable is executed. This syllable saves the current contents of ACVR, CVR, FVR and IVR and reloads them with the values of address of LCV, CV, FV and IV respectively which are on TOS. Finally, it causes capture of the PC as the address (word and byte) of the LRP (syllable following XCR).

The steps in the execution of XCR are

1. Pack and hold in the ARR (array result register) which is not in use at this time the following

	0 7	8 21	22 31
ICW1	IVR	FVR	ACVR

2. Store the contents of CVR at the address contained in the ACVR, i.e., CVR→(ACVR). This step is also available as an independent syllable PCV which will be discussed later.

3. Adjust the stack (ADJ)

   Integerize rA (INT)

   rA→CVR

4. ADJ, INT, rA→FVR

5. ADJ, INT, rA→IVR

6. ADJ, rA→ACVR

7. ARR→TOS

8. PC→TOS

Example:

```
|ACVR = CVR = FVR = IVR = 0|
FOR J = N + 2 TO Z BY -2
{ACVR = Address (J), CVR = N + 2
 FVR = 2, IVR = -2

 TOS = Address (LRPJ), | 0 | 0 | 0 |}
FOR I = 0 BY 4 TO J + X
{ACVR = Address (I), CVR = 0
 FVR = J + X, IVR = 4
 TOS = Address (LRPI), | -2 | 2 | A(J) |}
```

The net result of XCR is that the four values CV, FV, IV and address (LCV) are held in registers where they are easily accessed, and the immediately preceding values of these registers are either stored in memory (CVR) or held in the stack (IVR, FVR, ACVR) whence they may be used to restore ACVR, CVR, FVR, IVR when the inner loop is exhausted. The address of the LRP is also available in the stack for branching to the start of the loop when the inner loop is not exhausted. It should be noted, also, that the formulas used in computing CV, FV and IV can be quite general in that any non-integer result will be integerized (truncated) before it is used.

4.7.2 Accessing the Innermost LCV as an Operand

Once inside the body of a FOR-loop, the value of the LCV is kept in the CVR and the address of the LCV is kept in the ACVR. The value in CVR is incremented at the end of the FOR-loop but the value in the CVR is stored at (ACVR) only when necessary. Program events which require preserving the CVR are the following:

1. When entering an inner FOR-loop. This is taken care of automatically by the XCR syllable.

2. Before exiting to a subroutine. Once in the subroutine, the LCV may be sought by an ordinary FSS fetch from its memory location.

3. Before any GØTØ in which the label falls outside the range of the FOR-loop.

4. Before any switch GØTØ in which at least one label falls outside the range of the FOR-loop.

For the last three cases, the operator PCV (preserve current value) is used. The action is CVR→(ACVR).

Inside the innermost FOR-loop, the LCV is available by a 1 byte direct fetch from the CVR. This is the syllable FCV (fetch current value). The SCV (set current value) operation can be used to alter CVR.

```
PCV:  CVR→(ACVR)

FCV:  CVR→TOS

SCV:  TOS→CVR

FOR J = ————

                  | FSS I    gets I
                  | FCV      gets J
FOR I = ————

                  | FSS J    gets J
                  | FCV      gets I
END

                  | FSS I    gets I
                  | FCV      gets J
END
```

It should be noted that the two loop words put in the stack by XCR must be purged if the loop is aborted by a GOTO or a TEST statement. The restriction on entering a FOR-loop from anywhere but its top or by a subroutine return is absolute.

4.7.3 Termination of a FOR-loop

There are two syllables which can be used to terminate a FOR-loop.

The first of these is INX (increment index) which causes the following

1) IVR + CVR ⟶ CVR
2) Jump to the LRP address (rel to PSB) at TOS without deleting same.

This form is intended for case (b) of the concise loop statement.

The other loop terminating syllable, NDX, is for cases (a), (c) - (f) and accomplishes the following:

1) IVR + CVR ⟶ CVR
2) IF IVR $\geq$ 0 AND CVR $\leq$ FVR do step 2 of INX.
   IF IVR < 0 AND CVR $\geq$ FVR do step 2 of INX.
3) Otherwise
   3.1 Delete LRP address from TOS
   3.2 Restore registers as follows:

TØS (0//8) → IVR

TØS (8//14) → FVR

TØS (22//10) → ACVR

(ACVR) → CVR

If the terminated loop was nested, the restored contents of the loop control registers is the same as upon entry to the terminated loop and further computation or the NDX for the outer loop can occur.

4.8 Program Segmentation & Transfer of Control

A program segment is a block of control stream syllables of ≤ 1024 words with the first syllable on the word boundary (byte 0). To this may be added up to 1023 words of local scalar and array constants or non-control stream literals (non-integer or > 8191 in magnitude).

The first word of a program segment is marked by the program segment base (PSB) register which is reset by subroutine calls and inter-segmental jumps. Within a segment, data, in common with all data references, is addressed on the word boundary as one or two word items. These data items are addressed by the PSB relative fetching addressing operators FSP (single precision) and FDP (double precision) of which the relative 10-bit addresses are backwards relative to PSB, i.e., FSP 4 fetches the single word item whose address is PSB-4. By contrast, intrasegmental address references for transfer of control in the control stream are byte addresses, i.e., they are twelve-bit addresses of which the first ten address the word containing the byte and the last two address the byte (0-3) within that word. These addresses are forward relative to the PSB. Thus, a jump to address 102.1 transfers control to the second byte of the word stored at PSB + 102.

A FSP or FDP 0 will place a single or double precision 0 in the TOS.

4.8.1 Transfer of Control

We shall first categorize transfers of control as intersegmental and intrasegmental and then show how these are accomplished in the context of particular transfer operations and SPL program forms.

4.8.1.1 Intersegmental Transfers

Intersegmental transfers of control are of two types. The first is the subroutine call initiated by the ESP descriptor call. This is discussed separately in Section 4.9. We note here, only that an ESP initiated transfer is always to byte 0 of the first control stream syllable word of the new segment.

By contrast, the JOS (jump out of segment) descriptor call is not a subroutine jump (doesn't store return information) and causes a jump to an arbitrary byte address in the control stream of another program segment. The purpose of the JOS intersegmental jump is to permit a program to be of arbitrary length. The programmer and/or the compiler is not forced to chop a very long program up into a group of subroutines in order to comply with the 1024 word restriction on program length.

The JOS syllable is, as noted above, a descriptor call. The descriptor that it calls is called a program reference descriptor (PRD). A PRD has the following format:

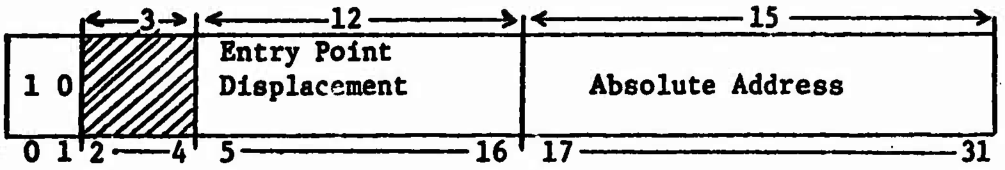


Figure : Program Reference Descriptor (PRD) Format

The steps in the execution of a JOS syllable are as follows:

1. JOS Q : (Q Rel PCT)→DDR

2. DDR (17//15) (address)→PSB

The previous contents of the PSB are lost.

3. The entry point displacement (epd) consists of 10 bits of word address and 2 bits of byte address. The word part is added to the absolute address:

3.1 DDR (5//10) + PSB→NS register

The byte portion is put in the next byte register

3.2 DDR (15//2)→NB (1//2)

0 →NB (0//1)

3.3 $(NS) \rightarrow IR_0$

$(NS+1) \rightarrow IR_1$

The instruction fetch cycle may now be initiated. The meanings of registers NS, NB, IR_0, IR_1 and the steps of the instruction fetch cycle are all found in Section 4.3 In that section, the abbreviation for steps 3.1, 3.2

Address + epd → PC

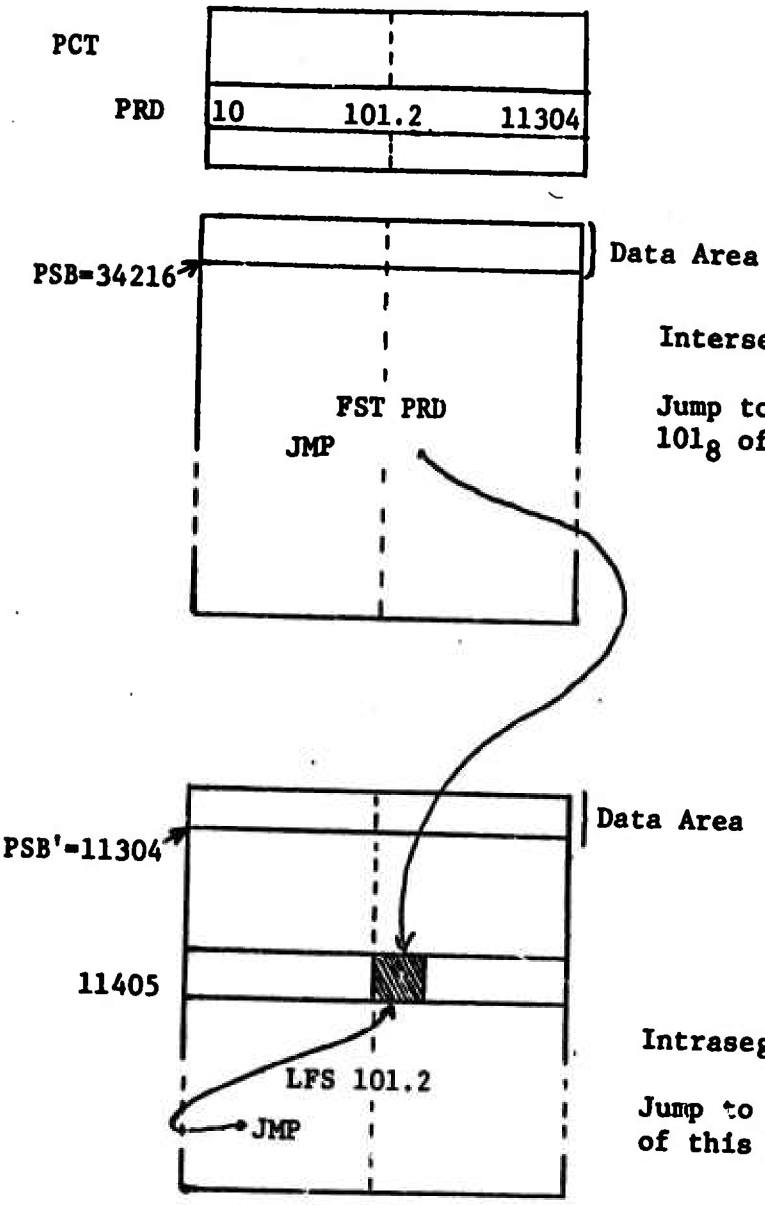
is explained. This abbreviation will be used in the balance of this section.

Intrasegmental Transfers

It may be noted that the JOS could also be used for an intrasegmental transfer but it takes six bytes to do so (4 bytes for the PRD, 2 bytes for the descriptor call). A more economical method as well as one offering the flexibility of conditional transfers is provided by the operators JMP (jump unconditionally), JOF (jump on false) and JOT (jump on true). These may be exercised by fetching a positive 12-bit literal to TOS (2-byte LFS) and then invoking one of these operators (1 byte) for a total of three bytes.

A compilation condition which arises in SPL (which does not require all labels to be declared at the beginning of a block) is the problem of forward labels, i.e., a label is used but it (the label) is not found until much later in the program. Thus, one has to generate an LFS, JXX pair and put the address of the LFS syllable on a list of unfulfilled addresses. When the whereabouts of the label becomes known, this list may be used to complete the LFS. Since we are allowing simple jumps between program segments, it may turn out that the missing forward label is in another segment. The simplest solution to this problem is to generate a PRD for the forward label, store it in the PCT and replace the LFS (2 bytes) with an FST of the PRD (also 2 bytes). This makes sense only when the following test is added to the execution of a JMP, JOF or JOT syllable:

- if the word at the TOS is really a PRD (i.e., is less than 0), put it in the DDR and if (JOF, JOT) the condition is met, behave as though a JOS had been executed.



Intersegmental Jump:

Jump to byte 2 of relative location 101_8 of segment starting at 11304.

Intrasegmental Jump:

Jump to byte 2 of $101_8^{\underline{st}}$ location of this segment.

A second software problem arises from the explicit declaration of a switch. An empty position in the declaration means that control should step thru if the switch index takes a value corresponding to the empty position. Since the switch may be used in many different places, the step-thru positions of the vector of labels stored in the PCT generated by the switch declaration must be filled with a "universal step-thru label". This may be accomplished by appending the following test behind the previous addition:

- if the address in the TOS is not a PRD, but the field (1//19) ≠ 0 then step thru.

An overall flowchart of a possible execution sequence for the three operators JMP, JOF and JOT is shown in Figure 4-10.

Example:

DECLARE SWITCH, SEVEN = (L0,,,L3, L4, L5,)

This declaration might cause the following switch vector to be generated and stored in the PCT at location 4772

4772	L0
	Step-thru
	Step-thru
	L3
	PRD L4
	PRD L5
	Step-thru

The array descriptor of this vector stored at PCT relative address 412:

0 1	0	▨	6	0	4772

Compilation of GØTØ SEVEN(K):

Length	Mnemonic	Address	Comment
2	FSS	K	
2	ELF	PCT412	Kth element of switch →TOS
1	JMP		

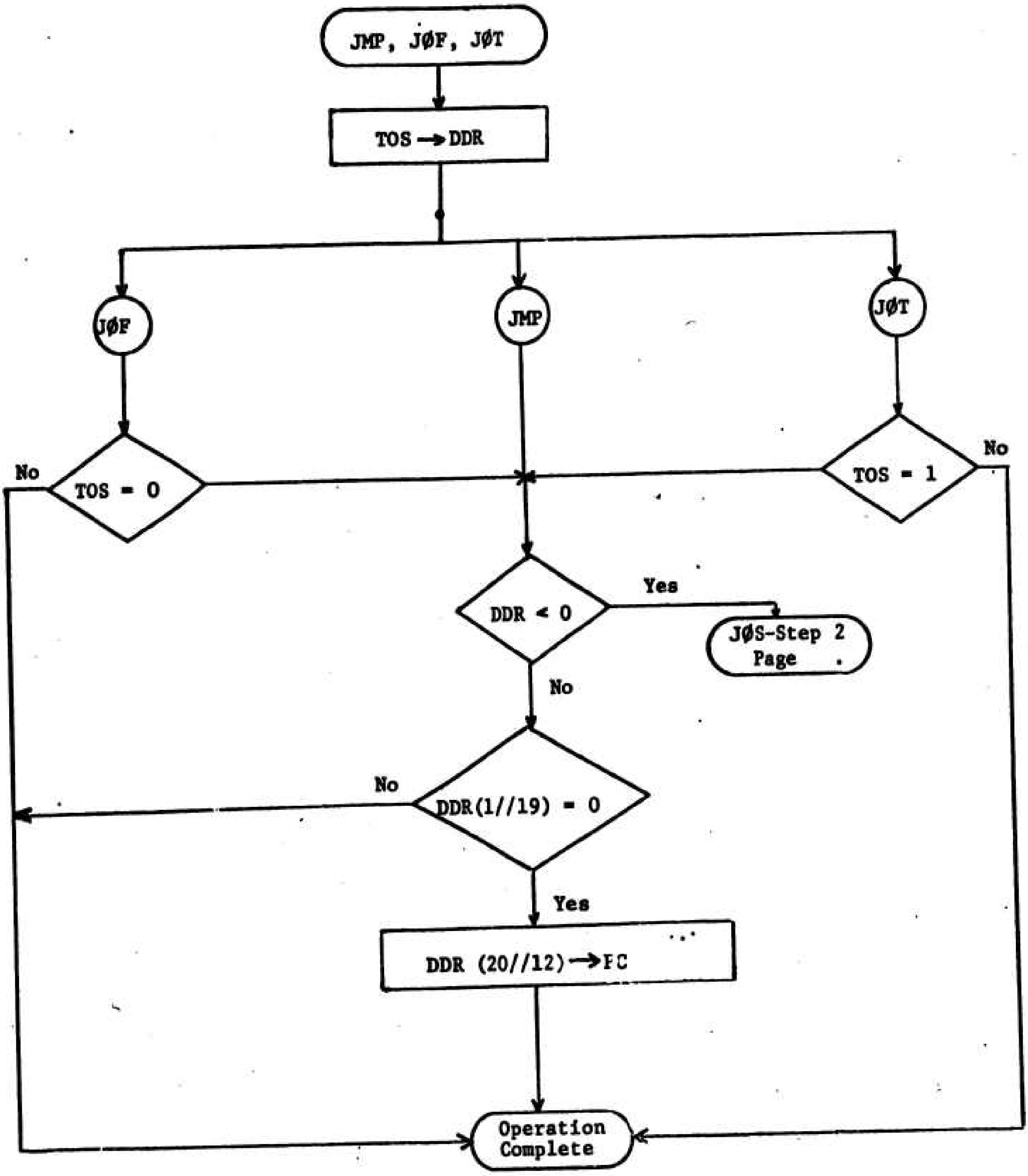


Figure 4-10: Execution of Transfer Operators JMP, JØF, JØT

4.9 Procedure Entry & Return

4.9.1 Setting Up A Calling Sequence

All the input and output arguments of a procedure are stored in the stack. The representation of an argument is either by value, by address or by descriptor according to the argument type and whether it is an input or output argument. The stack representations of the various argument types are set forth in Table 4-9.

The computational significance of storing the arguments in the stack is great efficiency in compiling and executing complex arithmetic statements in which there are nested calls to function procedures. For example:

```
Y = .SQRT (A**2 + (B*.SIN (X-.SQRT (.COS(Z))))))
```

compiles rather easily and compactly to

Step	Length	Mnemonic	Address	Comment
1	2	FSS	A	
2	1	DUP		
3	1	*		A^2
4	2	FSS	B	
5	2	FSS	X	
6	2	FSS	Z	Z, X, B, A^2
7	2	ESP	D(.COS)	COS Z
8	2	ESP	D(.SQRT)	$\sqrt{\text{COS Z}}$

Step	Length	Mnemonic	Address	Comment
9	1	-		X - COS Z
10	2	ESP	D(.SIN)	SIN(X - COS Z)
11	1	*		B* SIN (X - COS Z)
12	1	+		A^2 + B* SIN (X - COS Z)
13	2	ESP	D(.SQRT)	$\sqrt{A^2 + B* \text{SIN} (X - \text{COS } Z)}$
14	2	SRC	Y	Result → Y
	23			

A second consequence of using the stack to hold the arguments is that because the return information is also stored in the stack, a procedure with only input parameters represented by value can be called recursively. Thus, for example, the following is a valid SPL/MK III procedure for the AGC:

```
PROC .FIBON(N) I DECLARE I, N
IF N LS 0 THEN RETURN (.FIBON(-N)) ELSE
IF N LS 2 THEN RETURN (1)          ELSE
                     RETURN (.FIBON(N-1) + .FIBON (N-2))
ENDALL
EXIT
```

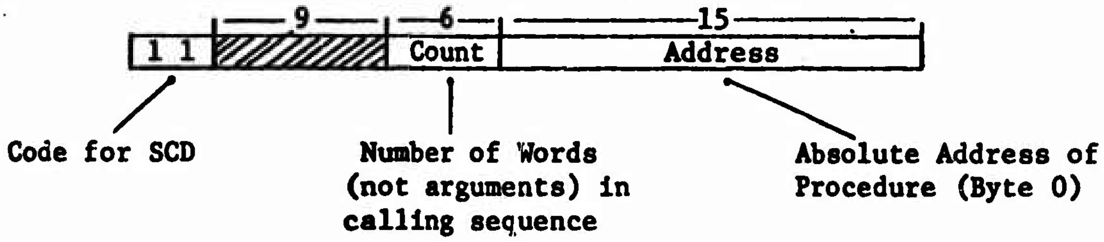
Implied in both of these examples is that a procedure which is used as a function leaves its (one or two-word) result on the TOS. This precludes the use of vector or matrix valued functions.

Actual Argument Type	Stack Representation	CSB Relative Addressing Operator or Descriptor Call for Access
INPUT		
1. Scalar		
1.1 Literal 1.2 Variable 1.3 Formula	Actual Value	FSC or FDC (See Section 4.5)
Latter two may include array elements.		
2. Array		
2.1 Whole Array (no formulas)	Array Descriptor (ARD)	AFC, AFR, ELF, XPF
3. Procedure Name	Subroutine Call Descriptor (SCD)	ESP (this section)
4. File Name for Input	Input/Output Descriptor (IOD)	IN (Section 4.13)
OUTPUT		
1. Scalar	Address of Scalar Relative to SPM	FSC followed by a left arrow store operator. (Section 4.5
2. Array		
2.1 Whole Array	Array Descriptor	ASC, ESC, ESK
3. Statement Label	Usually a Program Reference Descriptor (PRD), but see text, this section.	FSC followed by Abnormal Subroutine Exit operator (ASX). (this section)
4. File Name for Output	Input/Output Descriptor (IOD)	OUT (Section 4.13)

Table 4-9 : Stack Representations of Subroutine Arguments

4.9.2 Calling a Procedure

After the calling sequence has been loaded into the stack, the descriptor call syllable ESP (enter subroutine program) is issued. The ESP syllable contains either a ten-bit address which is taken relative to the PCT for a procedure whose name is not itself a procedure argument at the place where it is used,or else a six-bit address taken relative to the CSB (calling sequence base register) for calling a procedure name passed as a descriptor in a calling sequence. (See Section 4.9.3 and Table 4.9.) The descriptor called by the ESP syllable is a subroutine call descriptor (SCD). The format of an SCD is:



The functions of the ESP syllable are

- save return information in the stack
- reset the CSB register so that items in the calling sequence may be accessed inside the new subroutine
- transfer control to byte 0 of the first program segment of the procedure.

In detail, the steps are:

1. Recognize ESP syllable and transfer SCD descriptor to the DDR (descriptor decoding register).

2. Create two return parameter words (RPW's) storing each at TOS:

   2.1 PSB current contents → RPW_1 (17//15)

   2.2 PC current contents → RPW_1 (0//17)

   2.3 RPW_1 → TOS

   2.4 CSB current contents → RPW_2 (0//11)

   2.5 DDR (10//7) → RPW_2 (25// 7)

   count field
   of the SCD

   2.6 RPW_2 → TOS

3. Reset program counter (PC) and program segment base (PSB) registers as follows

   3.1 DDR (17//15) → PC word portion (NS register)

   0 → PC byte portion (NB register)

   3.2 DDR (17//15) → PSB

4. Force rA and rB into the memory portion of the stack leaving rP pointing to the memory location of RPW_2.

5. Reset the calling sequence base (CSB) register as

$rP \rightarrow CSB$.

The result of this is that the CSB in any subroutine points to the memory location of RPW_2, CSB-1 to RPW_1, CSB-2 to the last argument word, etc.

6. $(NS) \rightarrow IR_0$

$(NS+1) \rightarrow IR_1$ enter instruction fetch cycle. In words, these actions are: load first two control stream words of the subroutine into the instruction register.

Example:

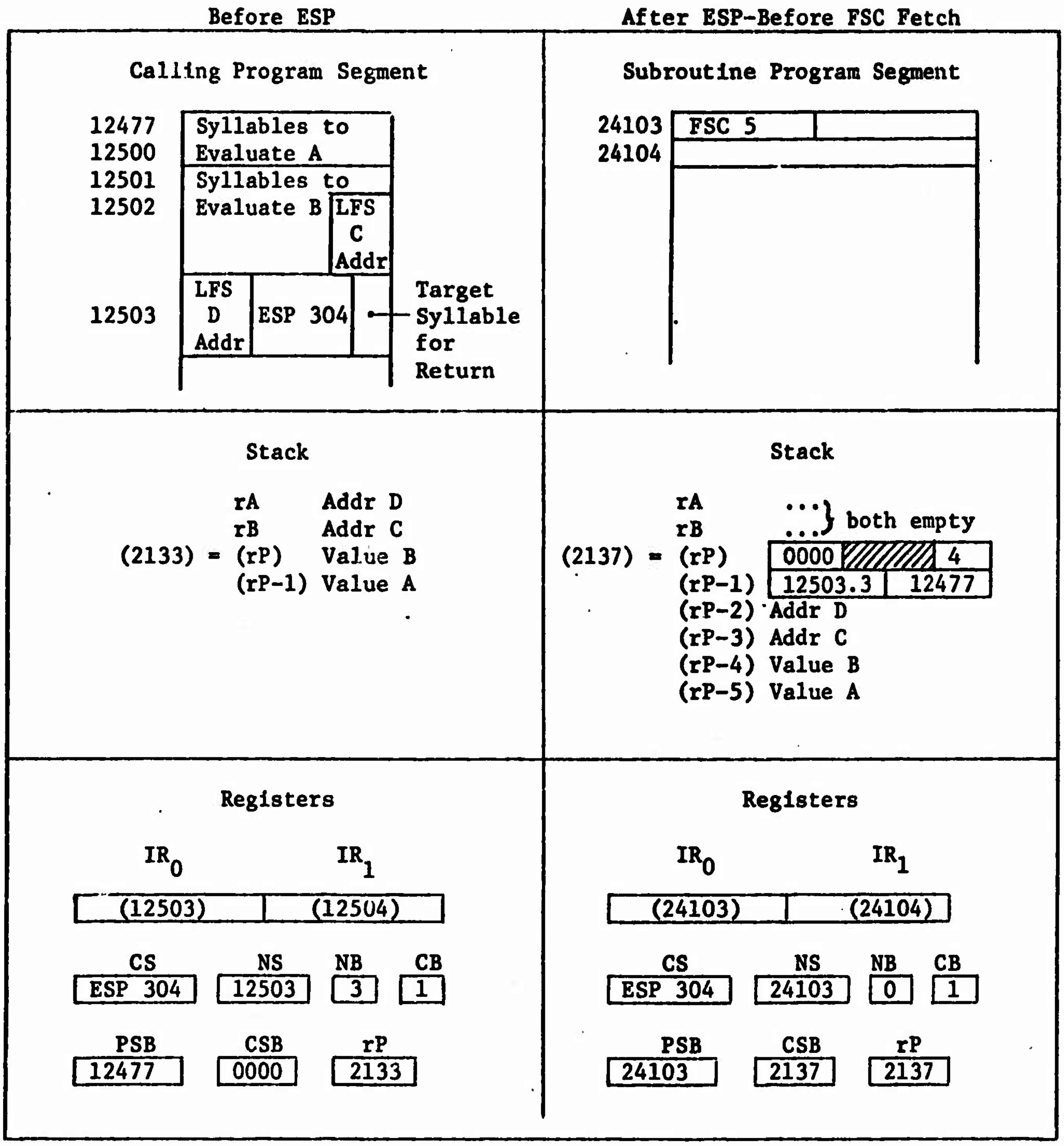
.SUBA (A,B = C,D)

Let us suppose that this subroutine is called at location 12503.1 of a program segment which began at 12477. We suppose, further, that A and B are scalar formulas; C and D are scalar items; that we are at the main program level so CSB = 0; and finally that the SCD for the subroutine is at PCT location 304:

1 1	▨	4	24103

SCD at PCT 304

i.e., the subroutine begins at byte 0 of 24103 and the subroutine starts, say, with a fetch of the first argument.


Before ESP
After ESP-Before FSC Fetch
Calling Program Segment
12477
12500
12501
12502
12503
Syllables to Evaluate A
Syllables to Evaluate B
LFS C Addr
LFS D Addr
ESP 304
Target Syllable for Return
Subroutine Program Segment
24103
24104
FSC 5
Stack
rA Addr D
rB Addr C
(2133) = (rP) Value B
(rP-1) Value A
Stack
rA
rB
... } both empty
(2137) = (rP)
0000
4
(rP-1)
12503.3
12477
(rP-2) Addr D
(rP-3) Addr C
(rP-4) Value B
(rP-5) Value A
Registers
IR_0
IR_1
(12503)
(12504)
CS
ESP 304
NS
12503
NB
3
CB
1
PSB
12477
CSB
0000
rP
2133
Registers
IR_0
IR_1
(24103)
(24104)
CS
ESP 304
NS
24103
NB
0
CB
1
PSB
24103
CSB
2137
rP
2137

4.9.3 Accessing Procedure Arguments

All procedure arguments are accessed with addressing operators or descriptor calls with a seven-bit address ($2 \leq$ address ≤ 65) which is taken relative to the CSB register. Thus, in the preceding example

FSC 5 fetches value of argument A

FSC 4 fetches value of argument B

FSC 3 fetches address of argument C

FSC 2 fetches address of argument D

A software consequence of passing arguments by value in the stack is that the requirement that the number and storage mode (single vs double precision) agree betwixt subroutine and calling program is absolute. If .SUBA, in this example, expects A and B to be double precision then, it will attempt to fetch the two words of A with FDC 7 (pulling up whatever was in the stack at the time the program began to generate the calling sequence). Seeking B, an FDC 5 will bring up the two single precision values A and B as though they were one double precision number. The addresses will be correctly fetched. Such an error has consequences also for the process of returning. This will be discussed later.

Running the relative addresses from 2 to 65 rather than 0 to 63 (seven vs six-bit CSB relative address) permits the programmer to have access to the RPW's. It should be noted that procedures are confined to a maximum of 64 words in their calling sequences. This count will not be the same as the number of arguments when any value argument is double precision.

Table 4-9 summarizes the addressing operators and descriptor calls used to fetch different argument types.

4.9.4 Return from a Procedure

4.9.4.1 Normal Return

The basic method for effecting a normal procedure exit is predicated on the notions that a procedure may be used as a function and that it is undesirable to have two different ways to return from a procedure, i.e., a subroutine exit and a function exit. Therefore, all procedure exits are done function style which means that the memory portion of the stack (rP) is returned to the position which existed just before the calling sequence was constructed ("winding down the stack") and the arithmetic registers are left holding any results which may have been generated by the procedure.

Example:

Figure 4-11 shows successive snapshots of the stack during the computation shown on page 4-78. The horizontal line divides the arithmetic registers from the memory portion of the stack. The steps numbers are keyed tc the step numbers shown on page 4-78. The stack pictures during the computations inside the functions called are not shown. The portions of the stack which must be removed during the procedure return process are boxed. The RPW's are shown as R_1 and R_2 with appropriate distinguishing superscripts.

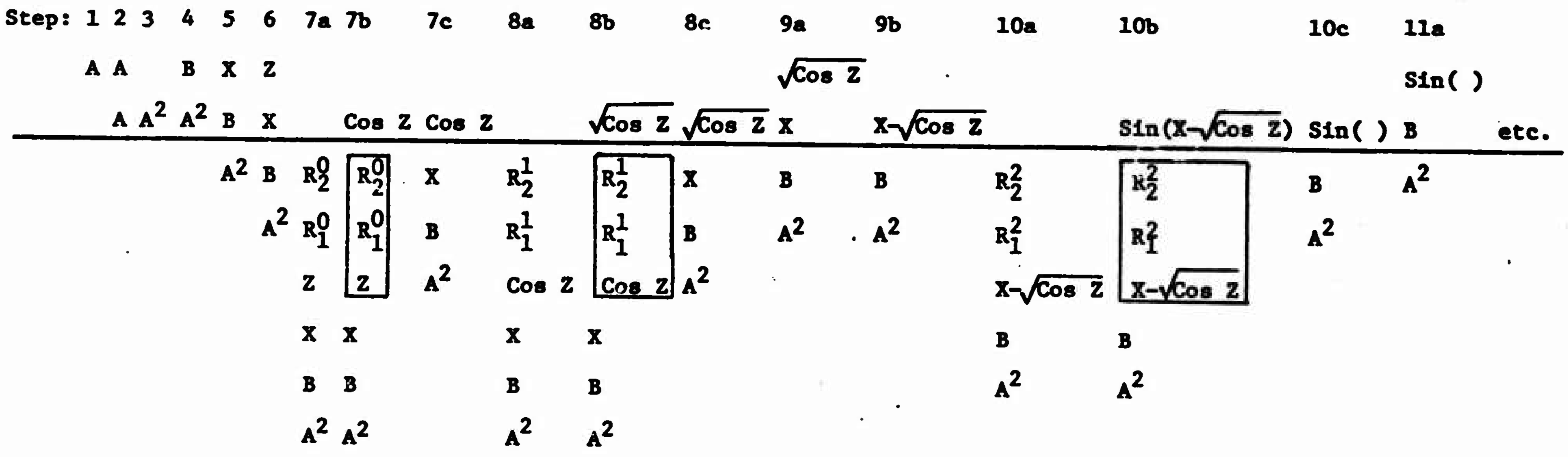

Step: 1 2 3 4 5 6 7a 7b 7c 8a 8b 8c 9a 9b 10a 10b 10c 11a
A A B X Z
$\sqrt{\text{Cos Z}}$
Sin( )
A A^2 A^2 B X
Cos Z Cos Z
$\sqrt{\text{Cos Z}}$ $\sqrt{\text{Cos Z}}$ X
$X-\sqrt{\text{Cos Z}}$
$\text{Sin}(X-\sqrt{\text{Cos Z}})$ Sin( ) B
etc.
A^2 B R_2^0 R_2^0 X R_2^1 R_2^1 X B B R_2^2 R_2^2 B A^2
A^2 R_1^0 R_1^0 B R_1^1 R_1^1 B A^2 A^2 R_1^2 R_1^2 A^2
Z Z A^2 Cos Z Cos Z A^2 $X-\sqrt{\text{Cos Z}}$ $X-\sqrt{\text{Cos Z}}$
X X X X B B
B B B B A^2 A^2
A^2 A^2 A^2 A^2

Figure 4-11 : Illustration of Stack Unwinding in Procedure Returns.

The normal procedure exit is caused by the operator NSX which is issued any place in a procedure where the flow of control requires a return. The steps undertaken in the execution of this operator are:

1. Restore the program counter:

   RPW_1 (0//17) → PC

2. Restore the program segment base

   RPW_1 (17//15) → PSB

3. Wind down the stack

   CSB - RPW_2 (25//7) -2 → rP if CSB ≠ 0

   This refers to the count field of RPW_2 which originally came from the count field of the SCD of the procedure. The result is that rP will be set to the value it had before the calling sequence was put in the stack. This step assumes that the count field and the actual number of argument words are equal. See also Step 3.3 on page 4-90.

4. Restore the calling sequence base

   RPW_2 (0//11) → CSB

   The steps shown access RPW_2 and RPW_1 as (CSB) and (CSB-1) respectively, i.e., the stack need not have RPW_2 at the top of the memory portion at the time the NSX is issued although this is the usual case.

4.9.4.2 Abnormal Return

Abnormal return is provided for non-function (subroutine) procedures. The SPL linguistic form is RETURN (statement label) where the statement label appears on the output side of the calling sequence with a period postfixed.

The stack representation of the abnormal return statement label depends on whether the label is local to the calling program or was passed to the calling program in its calling sequence. In the first case, the compiler puts a PRD for the label into the calling sequence and in the second case, stores in the calling sequence the relative position of the second hand label, i.e., if the label is CSB relative 3 in the calling sequence of the calling program then it passes the label on as the literal 3:

<u>Outermost Level (0)</u>

```
                                            Level 1

.SUBA (X = .ØSCAR)          ------->        PROC .SUBA (Q = LBL.)
     .                                           .
     .                                           .
     .                                           .
     .                                      .SUBB (V, W = LBL.)
     .                                           |
OSCAR.                                           |
                                                 v
                                            Level 2

                                            PROC .SUBB (Z, D = ERR.)
                                                 .
                                                 .
                                                 .
                                            IF Z LS 2*D RETURN (ERR)
```

Here OSCAR., since it is a label local to the outermost level, will be represented by a PRD in the call to .SUBA, whereas LBL. will be represented by 2 in the call on .SUBB since 2 is the CSB relative position of LBL. in the formal calling sequence of .SUBA.

The purpose of this distinction is to permit the stack to be wound down to the level it was at prior to leaving the subroutine level which contains the abnormal return label.

The abnormal return is initiated by bringing the return parameter to the top of the stack with an FSC addressing operator and then issuing an ASX operator. The steps undertaken in the execution of this operator are:

1. If rA contains a PRD (rA < 0) go to step 3, otherwise to step 2.

2. RPW_2 (0//11) → CSB

   (CSB-rA) → rA

   Return to step 1.

   This winds down one level and resets rA to the stack representative of the label at the previous level. Eventually a PRD must be reached.

3.

3.1 PRD address + PRD epd → PC

3.2 PRD address → PSB

3.3 CSB-RPW_2(25//7) -2 → rP

3.4 RPW_2(0//11) → CSB

4.10 Array Mode Processing

Array mode descriptor calls are used to fetch and store whole arrays (one or two-dimensional) or array cross-sections (rows or columns of a rectangular array). Array mode operators are used to perform the usual array on array and scalar on array operations (see Table 4-11) on the entities thus fetched. Array operations do not presently allow for general formula capability, i.e., only couplet formulas are accommodated. If a scalar operand is involved, it must be a literal or a simple (unsubscripted) variable; it may not be a formula.

All array mode arithmetic is either single precision (which includes integers) or double precision floating point according to the following rule:

- If either operand is double precision, double precision operations are used. The precision of an array is carried as a bit in the array descriptor.

- Single precision operations generate double precision results which are rounded to single precision prior to storage if the receiver is single precision with the RØUND attribute.

Array operations are performed using three stacks with stack pointer registers P_0, P_1 and P_2. The stack pointer register P_0 is the same as P, the stack pointer register of the ordinary scalar operations stack.

There is, in addition, a third double-word arithmetic register, ARR (Array Result Register) used in array mode operations. The typical array mode operation proceeds as follows:

1. Array mode is entered (EAM). This causes the contents of the arithmetic registers, rA and rB, if valid, to be forced into the memory portion of the ordinary scalar operations stack. Thereafter, as long as array mode continues, the arithmetic registers rA, rB and ARR are not considered to be part of either stack 0 (P_0), stack 1 (P_1) or stack 2 (P_2).

2. The first operand of the couplet (scalar or array) is fetched to stack 0.

3. The second operand of the couplet (scalar or array) is fetched to stack 1.

4. The array operation is issued. This causes

   4.1 Clear ARR

   4.2 Stack 0 operand → rA.

   4.3 Stack 1 operand → rB.

   4.4 rA OP rB (or vice versa)

   optionally + ARR → ARR

   4.5 When intermediate result in ARR is complete

   ARR → Stack 2.

5. When entire result is ready in stack 2, result is stored with an array storage operation which is a descriptor call.

The selection of stacks in steps 2, 3, 4 and 5 is automatic but may be overridden for operations such as simple array assignment (A = B) and array exchange (A == B).

A scalar is fetched to stack 0 or stack 1 by a literal fetch or an addressing operator as in scalar mode except that when the scalar has been fetched a stack lock flip-flop (SLFF) is set for the appropriate stack to permit repetitive fetching of the operand to the arithmetic registers.

When the operand to be fetched is an array and that array is transposed (primed), attention must be given to whether the operand is to be fetched by row or by column. In Table 4-10, we show by a C (for column) or R (for row) how an array is to be fetched. X and Y are arrays while S is a scalar. Note that the result is always generated by column.

			OPERAND 2		
			Y	Y'	S
OPERAND 1	X	+ - ÷	C C	C R	C ▨
		*	R C	R R	C ▨
	X'	+ - ÷	R C	R R	R ▨
		*	C C	C R	R ▨
	S	+ - ÷	▨ C	▨ R	▨
		*	▨ C	▨ R	

Table 4-10: Column vs. Row Fetch in Array Operations

It is anticipated that arrays will be stored forward by columns and it should be noted that array fetch operations (Steps 2 and 3 above) which move forward element by element will result in the array being stored forward in its stack but with the last element the first accessible.

4.10.1 Array Mode Control Stream Syllables

The four types of syllables shown in Table 4-1 are also found in array mode.

4.10.2 Literal Fetch Syllable - Array Mode

Performs the same function as in scalar mode except that the literal is fetched directly to the memory portion of either stack 0 or stack 1 and the corresponding stack lock flip-flop (SLFF) is set. SLFF0 and SLFF1 are both reset when a store operation has been completed.

4.10.3 Addressing Operators - Array Mode

The addressing operators with $0 \leq g \leq 7$ shown in Table 4-7 are recognized in array mode except that the operand is fetched and locked as described under literal fetch syllable immediately above.

4.10.4 Operators - Array Mode

Note that because of stack locking as described previously, it is not necessary to distinguish between element-by-element operations that are array on array and those that are scalar on array (or vice versa).

The following tables list the array operators.

Mnemonic	Function
AE+	Array Element-By-Element Operations X_i OP $Y_i \longrightarrow$ Stack 2 OR X_{ij} OP $Y_{ij} \longrightarrow$ Stack 2
AE-	
AE*	
AE ÷	
MXM	Matrix Multiply - Can produce scalar (1 X 1) result. When this occurs, the FSR (Fetch Scalar Result) operation can be used to retrieve it. See Table 4-12.
VXP	Vector Cross Product. Each operand must be 1 X 3 or 3 X 1.

Table 4-11 : Array Arithmetic Operators

The dimensions of an array are held in special control registers during an array operation. These are used to govern the execution of the array operations. Some rules that pertain are:

1. For AE operations

   - If both SLFF0 and SLFF1 are 0 then the number of elements generated is = minimum of the products of the dimensions of the two arrays.

   - If either SLFF0 or SLFF1 = 1 (both cannot be) then the number of elements generated is = product of the dimensions of the array operand.

2. For MXM

   - The usual rule on commensurability of dimensions is applied and if violated, no operation is undertaken.

3. For VXP

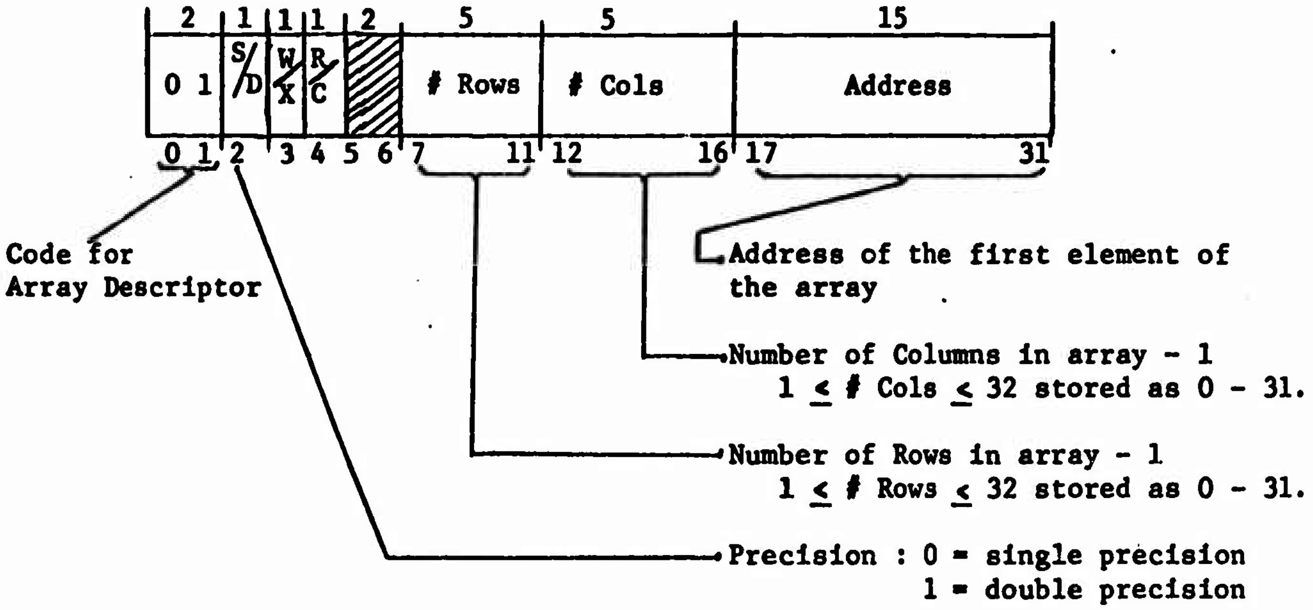
   - Each operand must be an array of dimensions 3 X 1 or 1 X 3.

Mnemonic	Function	
FSR	Fetch Scalar Result: Top element of Stack 2 → rA. Generally followed by ESM operation.	
SS0 SS1	Set Store from Stack 0 or Stack 1. These operations permit array assignment and exchange by permitting array storage from other than Stack 2.	
EAM ESM	Enter Array Mode Enter Scalar Mode	Same as in Table 4-6
FCV	Current Value of CVR → TOS - Same as in Table 4-6.	

Table 4-12 : Miscellaneous Array Operations

4.10.5 Descriptor Calls - Array Mode

Array mode descriptor calls fetch and store arrays or parts thereof where the array is described by the 32-bit array descriptor word addressed by the call. An array descriptor takes the form


2
1
1
1
2
5
5
15
0 1
S/D
W/X
R/C
# Rows
# Cols
Address
0 1 2 3 4 5 6 7 11 12 16 17 31
Code for Array Descriptor
Address of the first element of the array
Number of Columns in array - 1
1 ≤ # Cols ≤ 32 stored as 0 - 31.
Number of Rows in array - 1
1 ≤ # Rows ≤ 32 stored as 0 - 31.
Precision : 0 = single precision
1 = double precision

Bit 3: W/X = 0/1 for Whole Array vs. Cross-Section

Bit 4: R/C = 1 for Row vs Column Cross-Section if Bit 3 = 1

These two bits are not part of an ARD as stored in the PCT, but are created at object time by compiler supplied code and placed in the stack when an array cross-section is to be used as an argument to a subroutine. The address is also altered to the address of the leading element of the cross-section. Thus, the argument generated for A(,J) is

```
   (ARD(A) + J* (ARD(A)(7//5) + 1)((ARD(A)(1//2) + 1)) LOR
                                                  HEX'18000000'
```

Figure 4-12: Format of Array Descriptor (ARD)

The ARD is first put into the descriptor decoding register (DDR) and the S/D bit, the dimensions and the base address are captured. In the following i takes the value 0 or 1 depending on whether stack 0 or 1 is being loaded.

Register or Flip-Flop	Length	Function
$SPFF_i$	1	Stack Precision Flip-Flop receives S/D bit. This FF is also set by scalar fetches with an addressing operator or LFS.
BA_i	15	Base Address register receives base address.
M_i	5	$\left\{\begin{matrix}\#\text{ Rows}\\ \#\text{ Cols}\end{matrix}\right\}$ fetch by col; $\left\{\begin{matrix}\#\text{ cols}\\ \#\text{ rows}\end{matrix}\right\}$ fetch by row
N_i	5	
$SLFF_i$	1	Stack Lock Flip-Flop = 0 array operand = 1 scalar operand
P_i, i = 0, 1, 2	11	Stack Pointer register

Table 4-13: Array Mode Control Registers & Indicators

Depending on the particular descriptor call involved, these registers are then used in a microprogrammed loop to cause the desired result. As an example consider the descriptor call XPF (cross product fetch) used to fetch operands for a VXP operation.

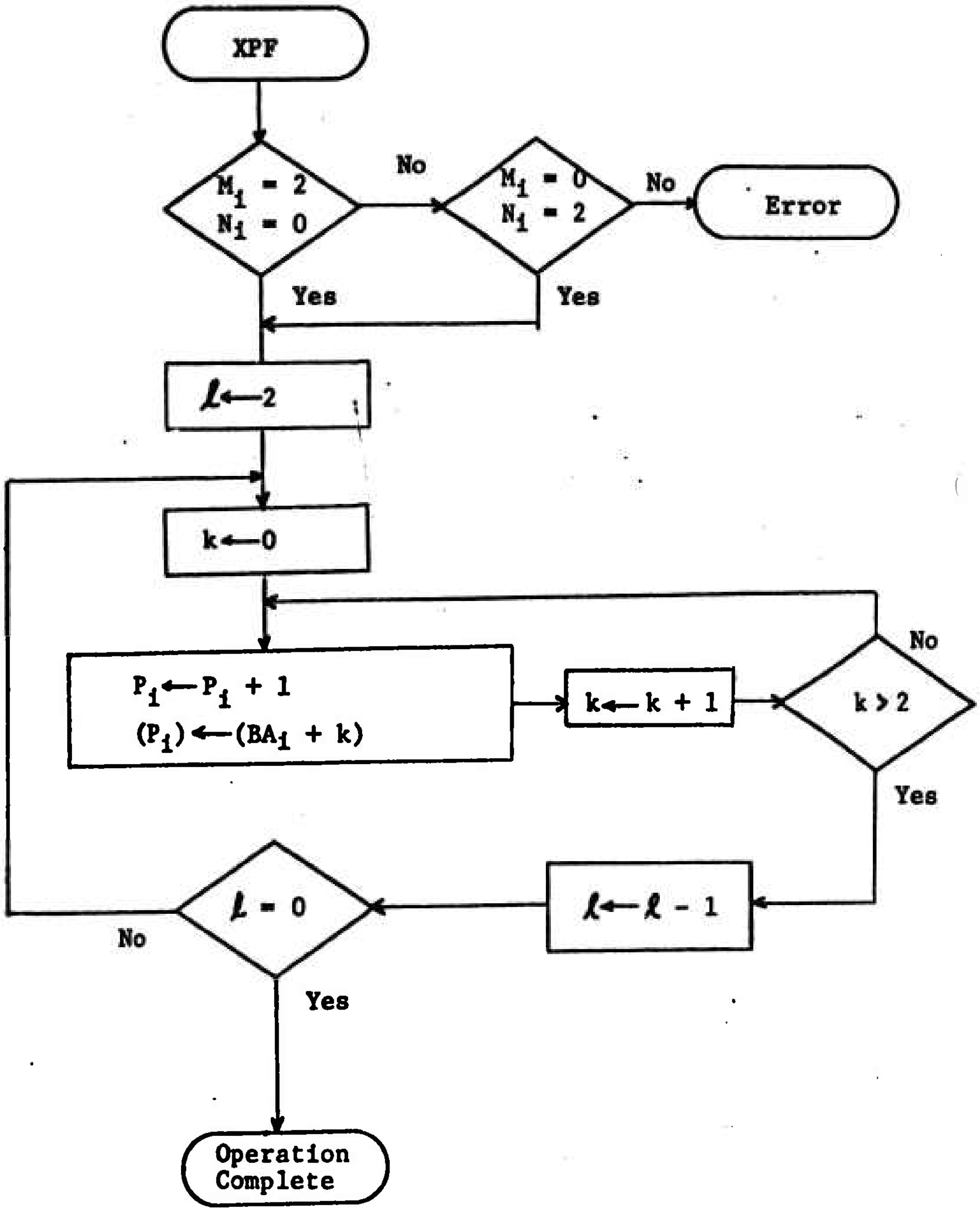


Figure 4-13: Flowchart of Cross Product Fetch (XPF)

The array mode descriptor calls are shown in Table 4-14. There are actually 16 such calls as each of the calls shown can address the PCT or the CSB as described in Section 4.5.4.

Mnemonic	Function
XPF	Cross-product Fetch: Fetches a 3-vector twice to the same stack.
AFC	Fetches an array by column to a stack.
AFR	Fetches an array by row to a stack.
$\left.\begin{matrix}\text{COX}\\ \text{ROX}\end{matrix}\right\}$	$\left\{\begin{matrix}\text{Column}\\ \text{Row}\end{matrix}\right\}$ X-section. Fetches the $\left\{\begin{matrix}\text{Column}\\ \text{Row}\end{matrix}\right\}$ whose subscript is on top of stack (literal or unsubscripted variable, no formula) to that stack deleting the subscript and turning off the SLFF which the LFS or the addressing operator turned on. A cross-section is a column or row of a rectangular array.
ASC	Store array by column from stack 2 unless an SS0 or SS1 has been invoked to cause storage from stack 0 or stack 1 respectively.
$\left.\begin{matrix}\text{SCX}\\ \text{SRX}\end{matrix}\right\}$	Stores the $\left\{\begin{matrix}\text{Column}\\ \text{Row}\end{matrix}\right\}$ X-section whose index has been fetched to the top of stack 2.

Table 4-14: Array Mode Descriptor Calls

The address portion of the array mode descriptor call refers to an address in the PCT or in thc subroutine calling sequence (CSB). This is indicated by the third bit of the call.

Examples:

1. Rotation of coordinates

```
DECLARE (3, 1) F 7, RBAR, RHAT

DECLARE, ROTATE (3, 3) F 7 = (8(0), 1)
     .
     .
     .
T1 = OMEGAE * TIR

ROTATE (0, 0) = -ROTATE (1, 1) = .SIN (T1)

ROTATE (0, 1) = ROTATE (1, 0) = .COS (T1)

RBAR = ROTATE * RHAT
```

Step	Length	Mnemonic	Address	Comment
(1	1	EAM		Enter Array Mode if not already in)
2	2	AFR	D(ROTATE)	Fetch Op 1 By Row → Stack 0
3	2	AFC	D(RHAT)	Fetch Op 2 by Col → Stack 1
4	1	MXM		Multiply
5	2	ASC	D(RBAR)	Store Result Stack 2 → RBAR
(6	1	ESM		Enter Scalar Mode if next statement is scalar)

7-9 one should also make note of the count in bytes of the descriptors (12) which in the worst case are not used elsewhere and the stack storage used (60).

Program	7- 9 bytes
Permanent Storage	12 bytes
Temporary Storage	60 bytes

The foregoing example can be better understood by inspecting snapshots of stack and register contents keyed to the step numbers of the example.

The following conventions are adopted:

R : Shorthand for RØTATE matrix

$\overline{R}$: Shorthand for RBAR vector

$\hat{R}$: Shorthand for RHAT vector

Subscript : Column number

Superscript : Row number

Vertical Arrow : A row or column with head at leading element.

Horizontal Arrow : Indicates target location of a stack pointer register.

For definiteness, let

$$R = \begin{bmatrix} 0.6 & 0.8 & 0 \\ 0.8 & -.6 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \hat{R} = \begin{bmatrix} 0.6 \\ 0.8 \\ 0 \end{bmatrix}$$

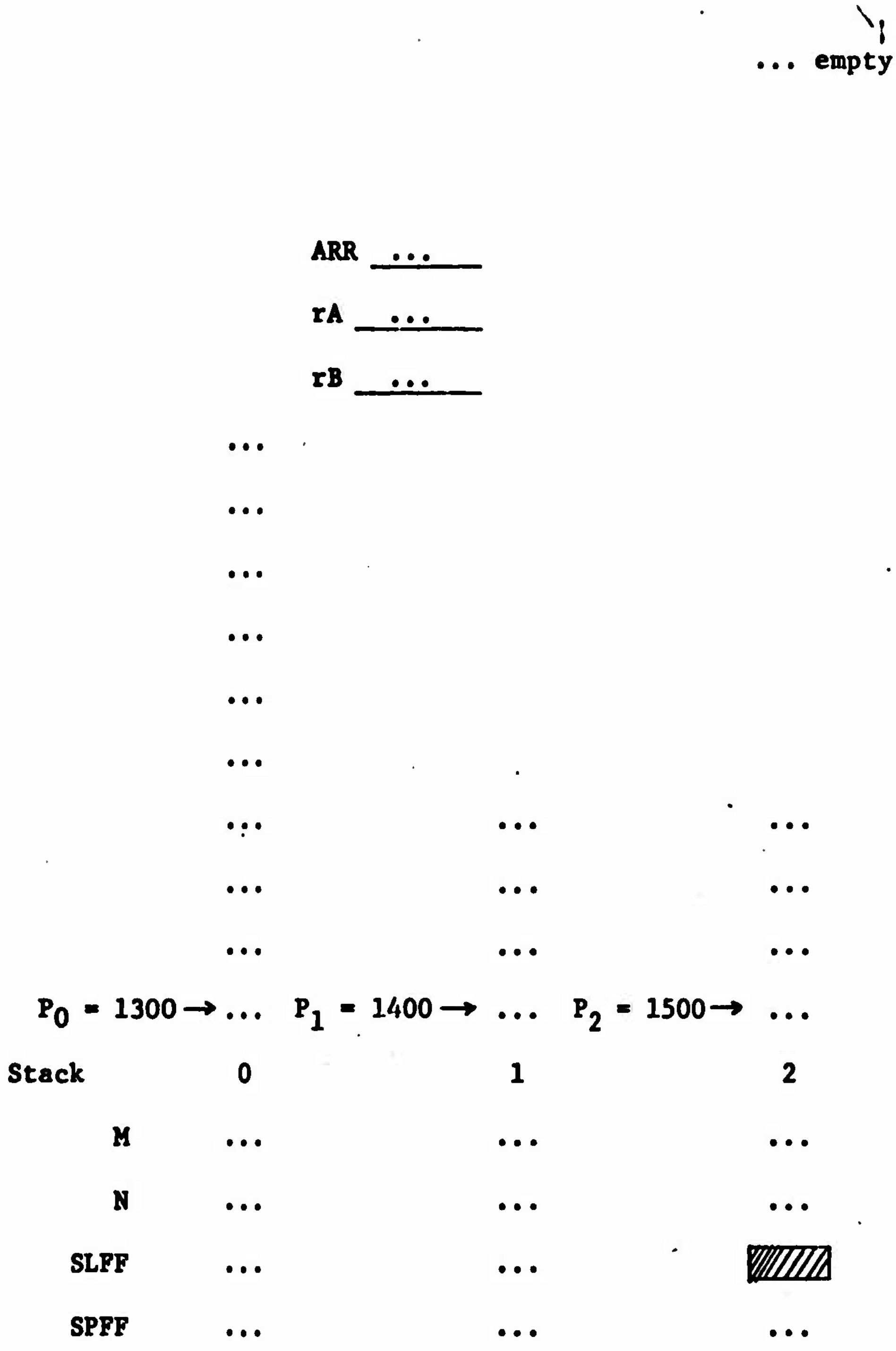


Figure 4-14a : Before Step 2

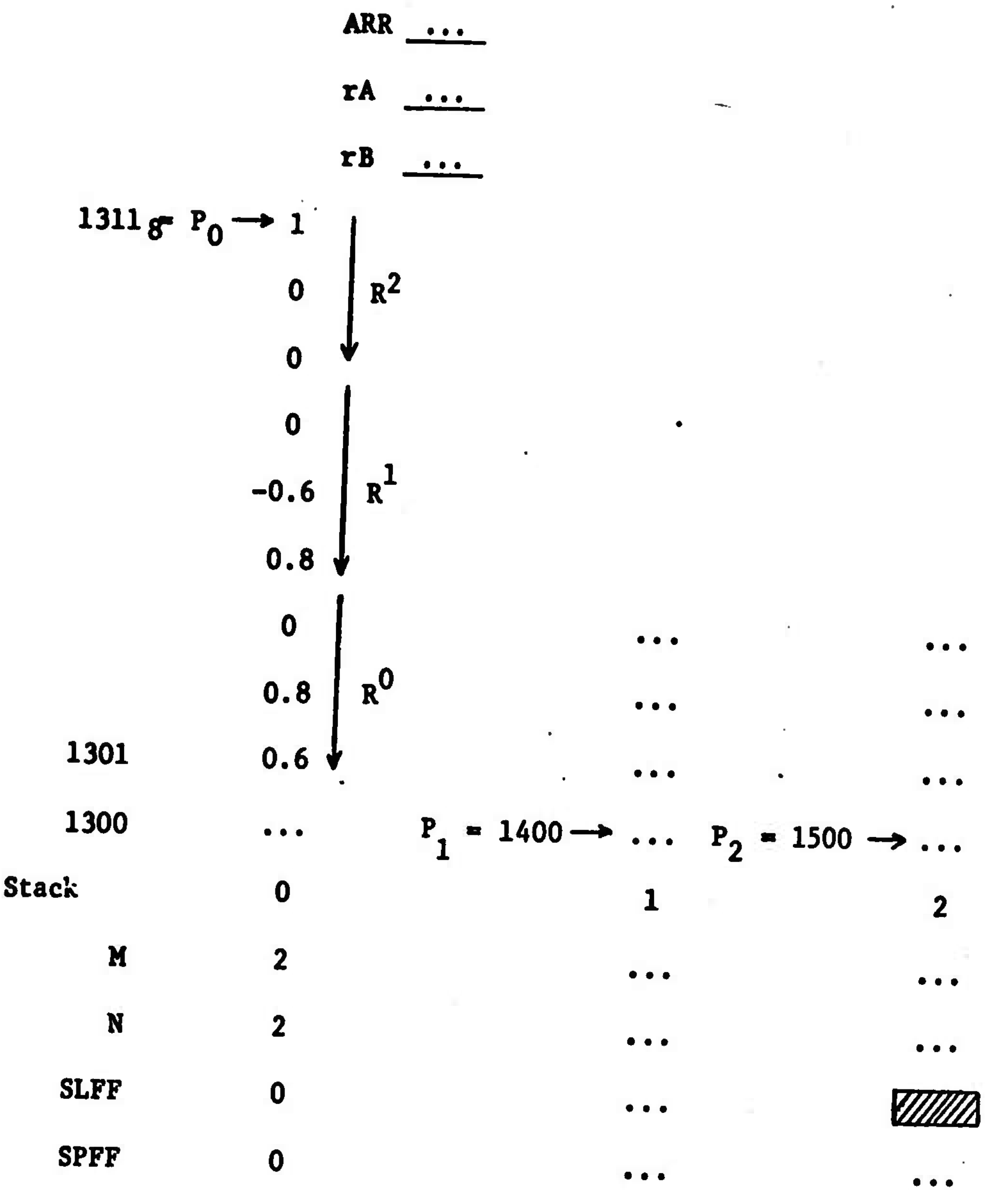


Figure 4-14b : After Step 2

ARR ...

rA ...

rB ...

1311_8 = P_0 →	1	R^2					
1310	0						
1307	0						
1306	0	R^1					
1305	-0.6						
1304	0.8						
1303	0	R^0	P_1 = 1403_8	0	$\hat{R}_0$		...
1302	0.8		1402	0.8			...
1301_8	0.6		1401	0.6			...
1300_8	...		1400	...		P_2 = 1500_8 →	...
Stack	0			1			2
M	2			2			...
N	2			0			...
SLFF	0			0			▨
SPFF	0			0			...

Figure 4-14c : After Step 3

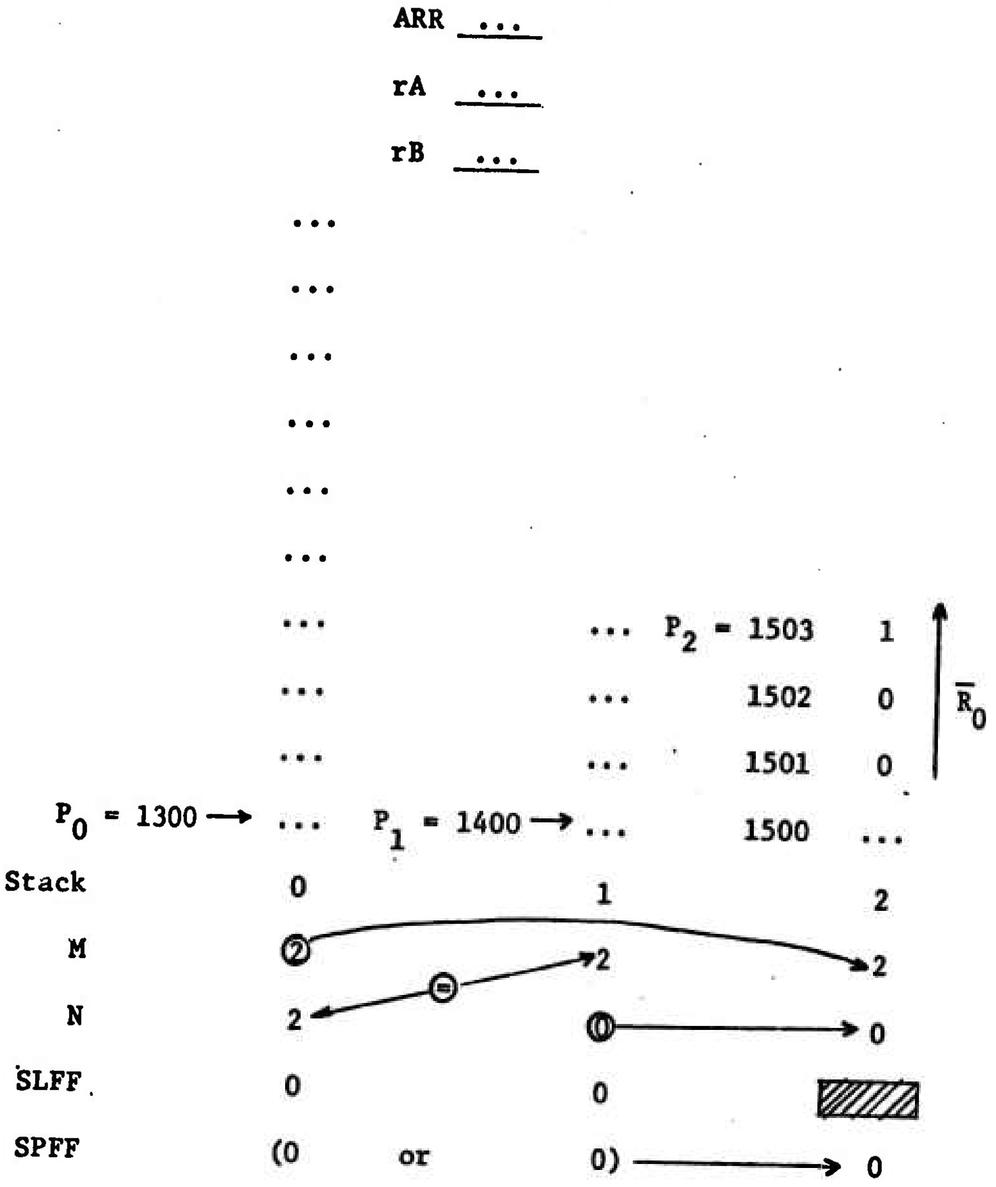


Figure 4-14d: After Step 4

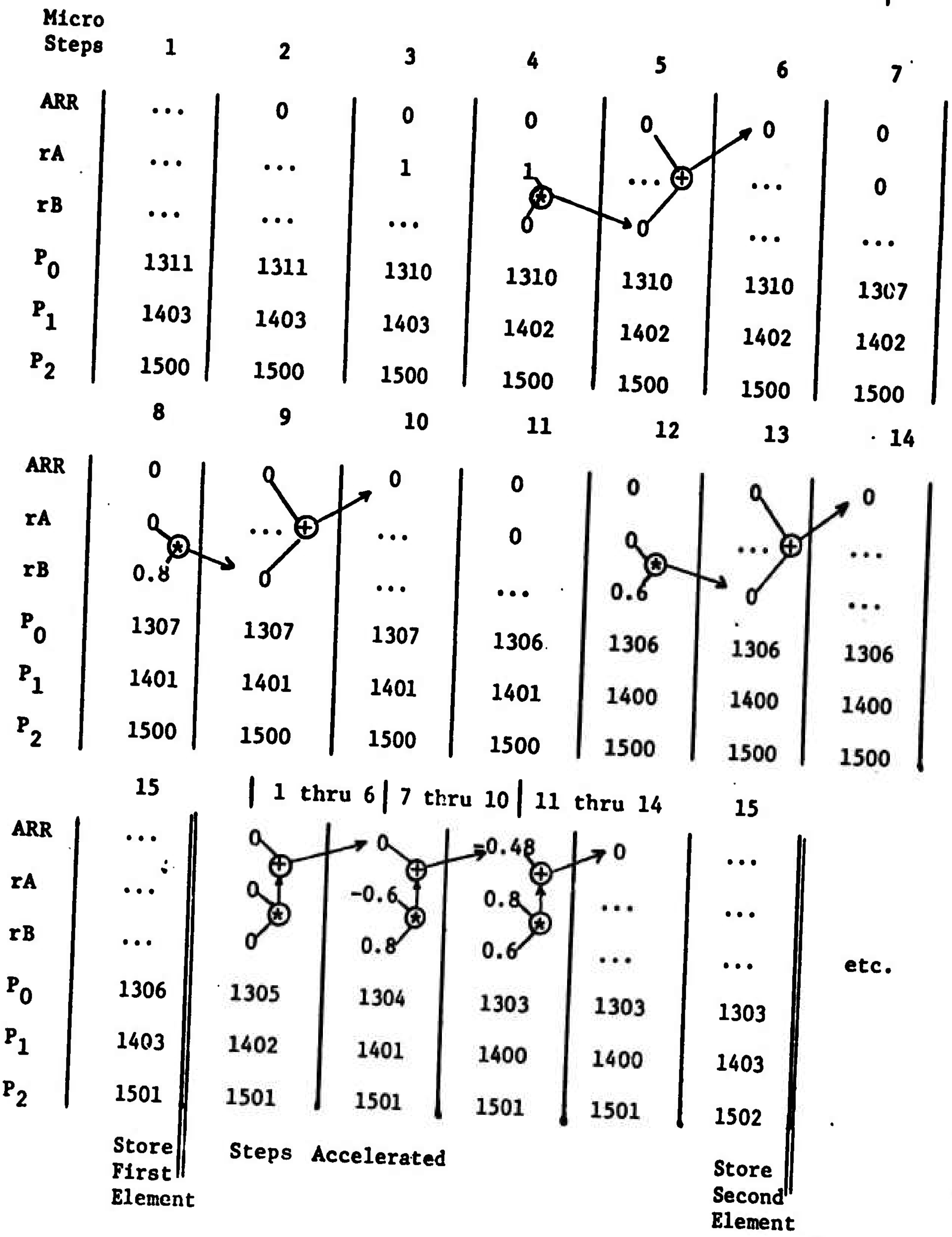
The snapshot which would succeed Figure 4-14d to illustrate the situation following Step 5 would be substantially the same as Figure 4-14a with the result $\overline{R}$ stored as follows

$$(1503) = 1 \rightarrow \overline{R}(0)$$

$$(1502) = 0 \rightarrow \overline{R}(1)$$

$$(1501) = 0 \rightarrow \overline{R}(2)$$

Of more interest is a dissection of Step 4 which appears in the following sequence.



Micro Steps	1	2	3	4	5	6	7
ARR	...	0	0	0	0	0	0
rA	...	...	1	1	...	...	0
rB	...	...	...	0	0	...	...
P_0	1311	1311	1310	1310	1310	1310	1307
P_1	1403	1403	1403	1402	1402	1402	1402
P_2	1500	1500	1500	1500	1500	1500	1500

	8	9	10	11	12	13	14
ARR	0	0	0	0	0	0	0
rA	0	...	...	0	0	...	...
rB	0.8	0	...	...	0.6	0	...
P_0	1307	1307	1307	1306	1306	1306	1306
P_1	1401	1401	1401	1401	1400	1400	1400
P_2	1500	1500	1500	1500	1500	1500	1500

	15	1 thru 6	7 thru 10	11 thru 14		15	
ARR	...	0	0	=0.48	0	...	
rA	...	0	-0.6	0.8	...	...	
rB	...	0	0.8	0.6	...	...	etc.
P_0	1306	1305	1304	1303	1303	1303	
P_1	1403	1402	1401	1400	1400	1403	
P_2	1501	1501	1501	1501	1501	1502	
	Store First Element	Steps Accelerated				Store Second Element	

Figure 4-15 : Detail Trace of Step 4

Example:

2. Set a whole Array to 0

Length	Mnemonic	Address	Comment
(1	EAM		Enter array mode)
1	LFS	0	Stack 0 has a locked-in 0
1	SSO		Set store from 0
2	ASC	D(A)	Store governed by dimensions in D(A)
(1	ESM		Enter Scalar Mode)
4- 6			

4.11 Interrupt System

The AGC interrupt system has been designed to permit

- reassignment of interrupt priority levels at the microprogram (firmware) level;
- enable/disable of individual interrupt conditions at object time;
- flexible respecification at object time action to be taken when an enabled interrupt occurs.

The following paragraphs describe the AGC interrupt system in greater detail starting with a description of its hardware/firmware/software infrastructure and then proceeding to a consideration of the compilation and execution of the appropriate SPL language elements.

4.11.1 Infrastructure of the Interrupt System

The interrupt system is based on hardware, firmware and software elements. The hardware elements consist of the following registers:

ICR: A 32-bit register (interrupt condition register) each bit of which corresponds to an interrupt condition. The particular bit corresponding to an interrupt condition is turned on (=1) when the condition occurs and is turned off (=0) when the condition has been satisfied.

IMR: A 32-bit mask register (interrupt mask register) each bit of which corresponds to an interrupt condition. When the IMR bit is on (=1) the interrupt condition is enabled and when it is off (=0) the interrupt condition is disabled (will not occur).

IFF: A flip-flop (interrupt flip-flop) which indicates whether (=1) or not (=0) an interrupt is presently in force, i.e., the logical AND of ICR and IMR ≠ 0.

IPR: A 4-bit register (interrupt priority register) which contains when IFF = 1 the priority level (0-15) of the interrupt in force. The lowest priority corresponds to IPR=15 and the highest to IPR = 0.

The ICR and IMR may be fetched and stored by operators provided for these functions (see Table 4-5). These operators are

FIC: TOS ⟵ ICR
SIC: TOS ⟶ ICR

FIM: TOS ⟵ IMR

SIM: TOS ⟶ IMR

It is not anticipated in actual operation that all interrupts should be maskable, but having the ability to do so can prove to be a material aid in debugging and simulation. The AGC compiler can be designed to forbid masking certain inputs depending on a parameter supplied to the compiler before compilation.

The firmware elements are the four priority field words (PFW's) which are stored in dedicated locations in the high-speed microprogram memory. Each PFW contains eight four-bit fields and each of the total of 32 such fields corresponds to one of the32 interrupt conditions. Each field contains the priority level (0 highest to 15 lowest) of the corresponding interrupt condition. In actual operationc the PFW's may be set by whatever means are eventually adopted for revising the microprogram logic control of the AGC processor.

The software element of the AGC interrupt system is a block of 16 words reserved in the (NDRO) PCT (program control table). Each word contains two interrupt transfer addresses (ITA's) wh. .h specify in the manner to be described later the entry points to the interrupt action programs.

4.11.2 Initiating and Returning from an Interrupt

The highest priority interrupt will interrupt at many places in the instruction fetch and execute cycle. The lower priorities will not take effect until the current instruction has been executed.

The steps involved in initiating an interrupt are illustrated in Figure 4-16. The steps contained in the last box of this flowchart are quite similar to the actions undertaken by the ESP syllable (Section 4.9):

1. Create two return parameter words (RPW's) storing each at TOS:

   1.1 PSB → RPW_1 (17//15)

   1.2 PC → RPW_1 (0//17)

   1.3 RPW_1 → TOS

   1.4 CSB → RPW_2 (0//11)

   1.5 IFF → RPW_2 (11// 1)

   1.6 IPR → RPW_2 (12//4)

   1.7 RPW_2 → TOS

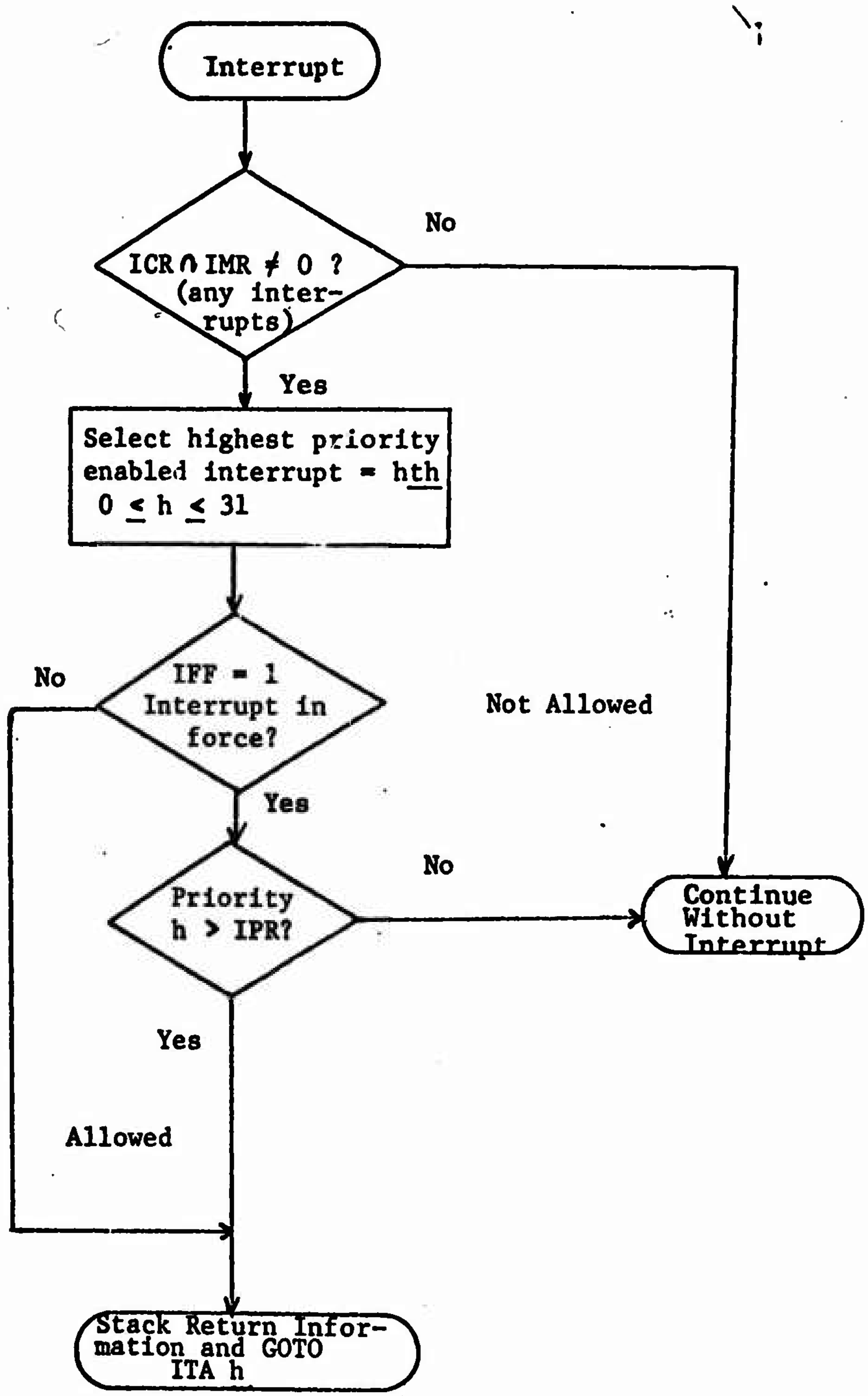

Interrupt
No
ICR ∩ IMR ≠ 0 ? (any interrupts)
Yes
Select highest priority enabled interrupt = hth 0 ≤ h ≤ 31
No
IFF = 1 Interrupt in force?
Not Allowed
Yes
No
Priority h > IPR?
Continue Without Interrupt
Yes
Allowed
Stack Return Information and GOTO ITA h

Figure 4-16: Initiation of Interrupt

2. Set up new interrupt indicators

2.1 1 → IFF

2.2 Priority h → IPR

2.3 0 → ICR (h//1)

Note that Priority h is a four-bit field from the four PFW's.

3. Reset PSB and PC as follows:

3.1 ITA → PC word portion (NS register)

3.2 0 → PC byte portion (NB register)

3.3 ITA h → PSB

Note that ITA h is a 15-bit field from the 16 ITA words.

4. Force rA and rB into the memory portion of the stack leaving rP point to the memory location of RPW_2.

5. Reset CSB as

rP → CSB

Thus, CSB relative addressing can be used in the interrupt action program to inspect the items that were in the stack at the time the interrupt occurred.

6. (NS) → IR_0

(NS+1) → IR_1 enter instruction fetch cycle.

The net result of this process is that interrupt return information is accummulated and stored in the stack and is available to restore the processor to its pre-interrupt status; this may be to non-interrupt level or to a lower level interrupt. (Note that a regular procedure can be invoked from an interrupt action program.)

The return from an interrupt action program is effected by the NSX operator. This functions in essentially the same fashion as the NSX procedure exit operator except for the additional restoration of the IFF and IPR from the appropriate fields of RPW_2 triggered by the on status of the IFF.

4.11.3 SPL Language Considerations

The SPL language considerations with which we are concerned here are those used to enable and disable (inhibit) specific interrupt conditions and to define the program action(s) to be undertaken when an enabled interrupt condition occurs.

4.11.3.1 Enable/Disable

Enable and disable of specific interrupt conditions is accomplished in SPL by the use of the UNLOCK (enable) and LOCK (disable) statements.

If, for example, the programmer wishes to disable the floating point underflow interrupt, he might write

```
DECLARE HARDWARE, FPU = ICR (5//1)
         .
         .
         .
LOCK     FPU
```

The declaration says that the programmer wishes to associate the sixth interrupt condition with the mnemonic FPU. The subsequent LOCK (UNLOCK) statement causes a 0 (1) to be stored as the sixth bit of IMR.

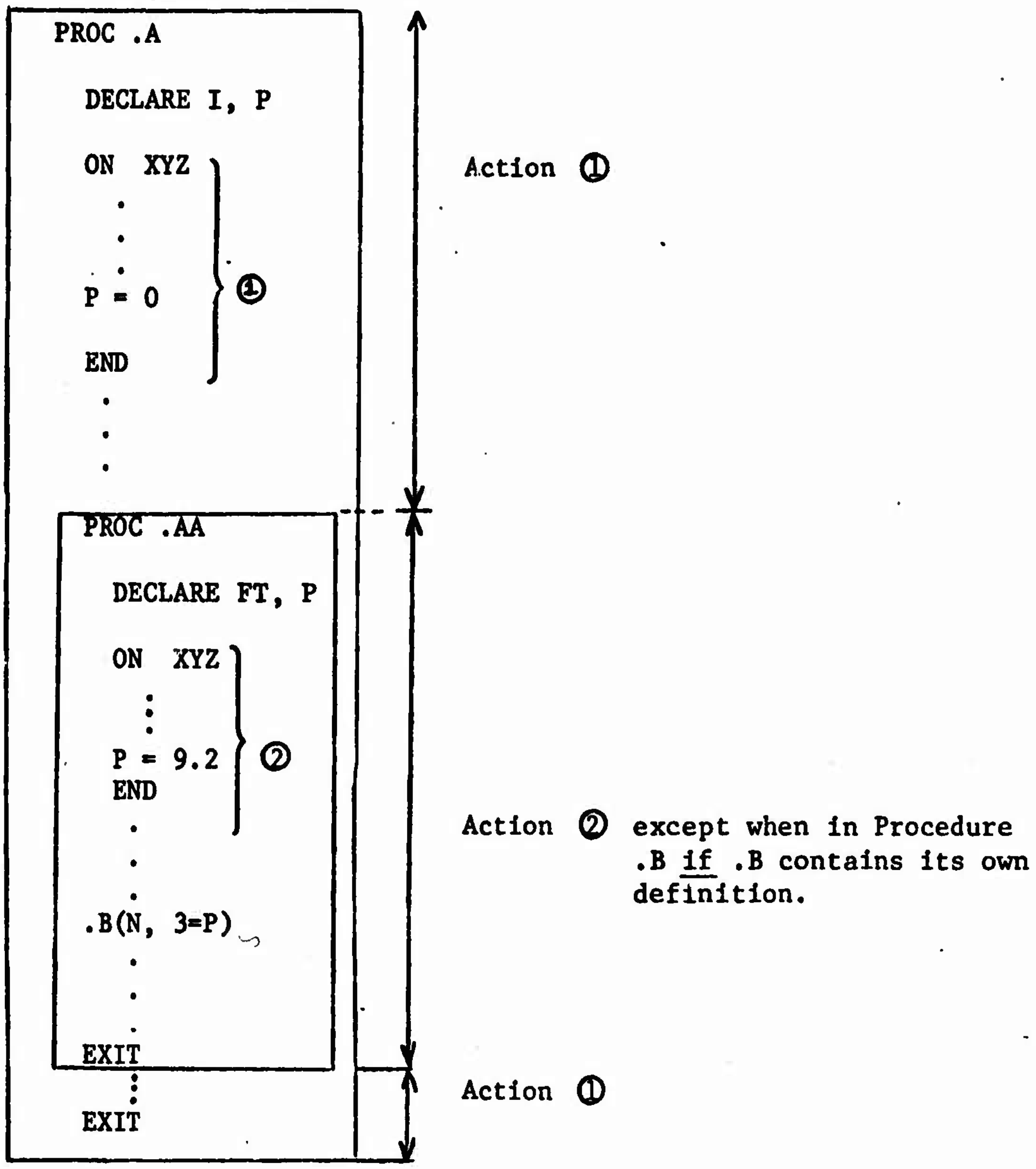
4.11.3 Specification of Interrupt Action

The specification of interrupt action must be discussed in terms not only of the compilation process but also in terms of the process whereby the independently compilable SPL/MK III procedures are linked and loaded.

The SPL language element for defining an interrupt action is the ON statement group. In terms of the above example,

```
ON FPU
Statement 1
  .
  .
  .
Statement n
END
```

Such a grouping may appear in the main program as well as in any procedure. The data definitions used in the statement group are those pertaining to the procedure in which the ON-statement group is embedded and the action defined by the ON-statement group is effective only in the execution scope of that procedure. This is clarified by the following illustration:

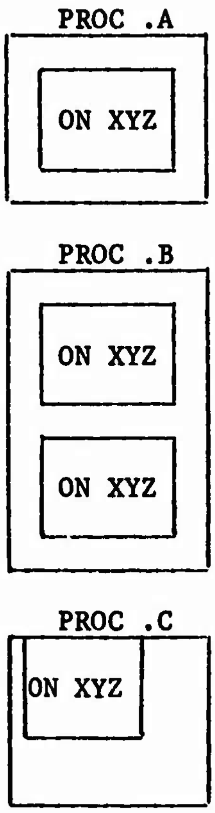


```
PROC .A
  DECLARE I, P
  ON  XYZ
     .
     .
     .
  P = 0
  END
     .
     .
     .
  PROC .AA
    DECLARE FT, P
    ON  XYZ
       .
       .
       .
    P = 9.2
    END
       .
       .
       .
  .B(N, 3=P)
       .
       .
       .
  EXIT
       .
       .
       .
  EXIT
```

except when in Procedure .B if .B contains its own definition.

The remaining facts of interest are that the ON-statement group is not executed at the point of its physical appearance and second, its effect begins at the point in execution order where it is physically located.

Let us suppose, as an example, that the loader encounters four ON-statement groups pertaining to the same interrupt.

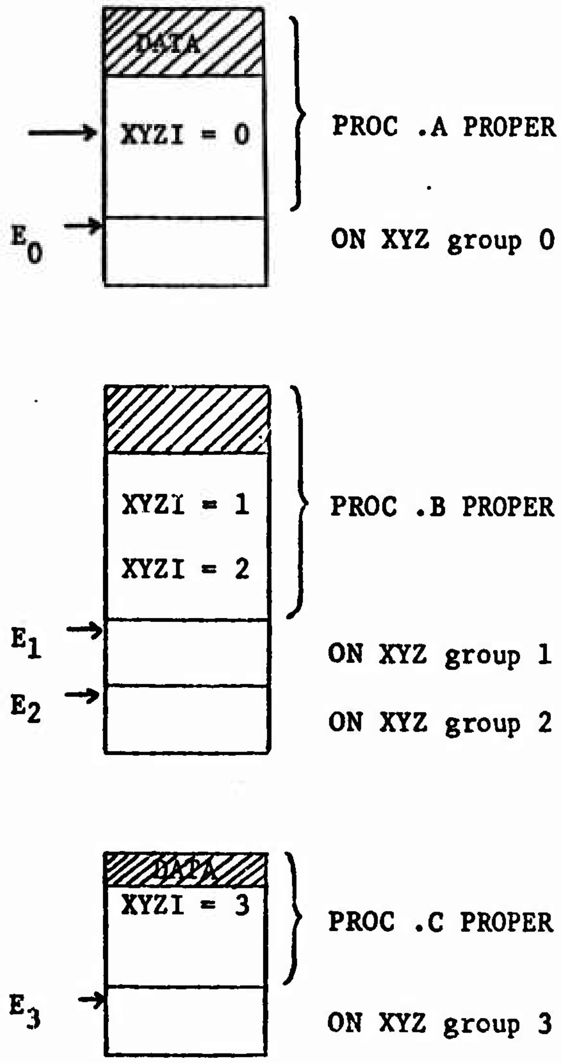


The compilation response is to compile each procedure, using separate

location counters for procedure control stream,
procedure local constants,
and for each ON-statement group control stream.

Creating a reserved variable name to pertain to the interrupt switch index (in this case call it XYZI), the same procedures ready for loading might appear

AT POSITION OF
ON XYZ group



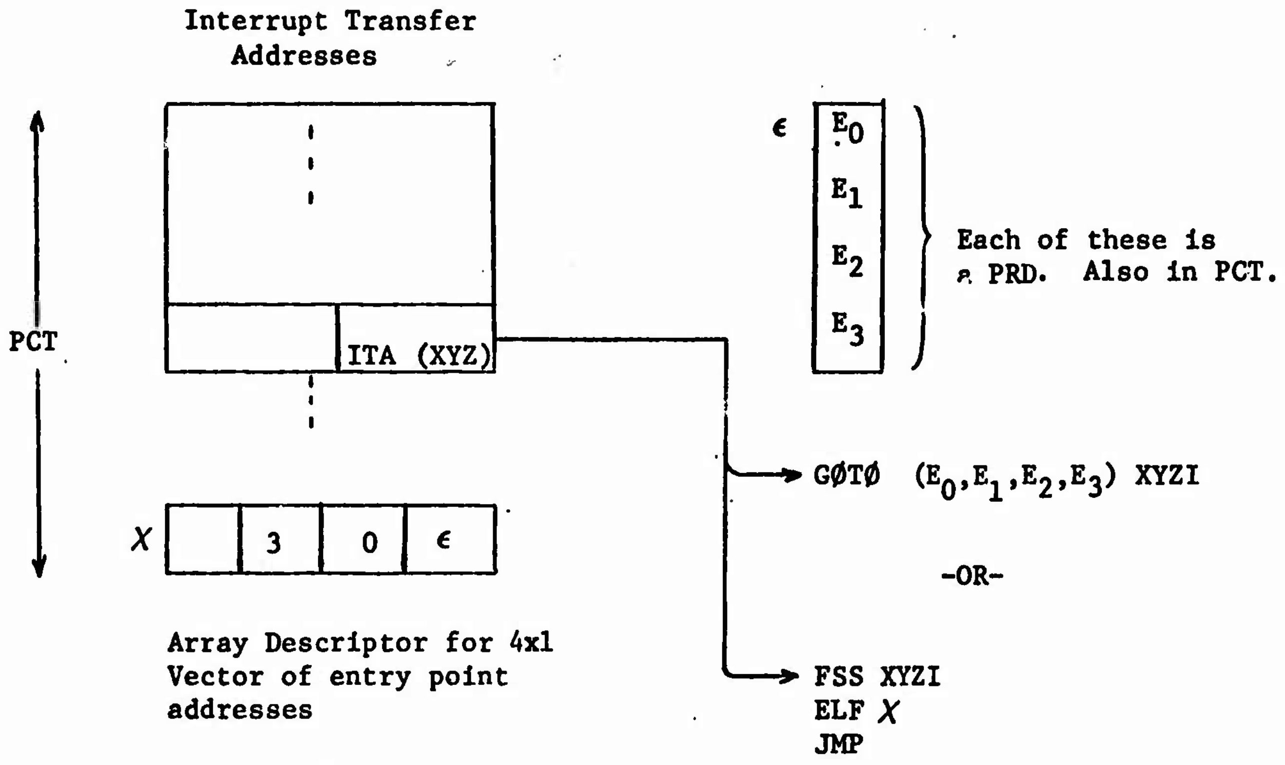
Notes:

- E_0, 1, 2, 3:

  Addresses of
  entry points to
  ON XYZ groups

- The values for the
  assignments to
  XYZI are filled
  in by loader.

The function of the loader is now to create a vector of the entry point addresses for each interrupt and include the code to use a switch based on that vector:


Interrupt Transfer Addresses
PCT
ITA (XYZ)
X
3
0
ϵ
Array Descriptor for 4x1 Vector of entry point addresses
ϵ
E_0
E_1
E_2
E_3
Each of these is a PRD. Also in PCT.
GØTØ (E_0,E_1,E_2,E_3) XYZI
-OR-
FSS XYZI
ELF X
JMP

The total process then is as follows: In executing the program, the physical locations of the ON-groups are encountered. These have been replaced by assignments of integer values (0-31) to a switch index variable (XYZI in this case). When the (XYZ) interrupt occurs, transfer is made to the loader generated code which starts at the address stored at ITA (XYZ). This code amounts to a switched GOTO on the index XYZI. This results in an unconditional transfer to one of the entry points E_0, E_1, E_2 or E_3. Each of the ON-groups programs ends (logically) with an NSX return to the point where the interrupt was detected and accepted.

If only one action definition for a given interrupt condition is encountered, the loader may set the ITA directly to its entry point and not generate a vector or additional coding. The two bytes required for the statement XYZI = 0 are backfilled with NOP's and XYZI is used for some other purpose.

4.12 Circumvention

The architectural organization permits circumvention to proceed in a very straightforward manner. At selected points in the real time program execution, such as at the completion of the real time interrupt routine or the minor loop, a program record (PRR) operator is executed as part of the executive routine. This operator causes the multiple storage of program word 1 (RPW1), program word 2 (RPW2), and Index Control word 1 (ICW1) which contain the status of the processor for a normal return from a subroutine or an interrupt routine (see sections 4.9 and 4.11), and the index control information for looping (see section 4.7). This status data is recorded in 9 dedicated scratchpad memory locations.

In addition the current value of the index is stored in its memory address (ACVR). The microprogram orders together with addresses to control high speed storage, are contained in 1024 words of program unalterable memory.

Circumvention is initiated by the highest priority interrupt upon sensing the indicated conditions. Upon termination of the condition the control logic reloads the high speed memory from the micro-logic and control addresses stored in the program unalterable memory, and then restarts the processor at the address stored in the program control table (PCT) for the highest priority interrupt (ITA_0).

The recovery routine which begins at the address stored in the PCT resets the critical output discretes, tests redundantly stored quantities, and corrects them where necessary, recovers elapsed real time from the system hardened timer and reconstructs or obtains accumulated velocity and platform reference. Finally, the recovery routine tests for the correct set of the multiply stored RDW1, RDW2 and ICW1 places them in the top of the stack, and returns control by issuing a program recover (PCR) syllable. The PCR syllable reloads the status of the processor from the top 3 words in the stack and transfers control to the address in the program counter.

The circumvention logic is recursive and permits the recovery routine to handle any number of circumvention conditions.

Example: PROGRAM RECORD

In Executive Routine

Control Stream		Stack		Index Registers	
PSB:	12047 →.	rA	...	ACVR	1057
	PRR	rB	...	CVR	3
PC:	13572.3 →...	(rP)	13051.1	FVR	9
		(rP-1)	+1 | 9 | 1057	IVR	+1
		CSB:	1237 → ——		

Result of PRR

Symbolic Name is RCØVER (3x3 array)

9 Dedicated Locations in Control Memory

13572.3	12047		RPW1 - Copy 1
1237	▨		RPW2 - Copy 1*
+1	9	1057	ICW1 - Copy 1
13572.3	12047		RPW1 - Copy 2
1237	▨		RPW2 - Copy 2*
+1	9	1057	ICW1 - Copy 2
13572.3	12047		RPW1 - Copy 3
1237	▨		RPW2 - Copy 3*
+1	9	1057	ICW1 - Copy 3

Memory: (ACVR) = (1057) = 3

*IFF, IPR and Count fields are not relevant to this function.

Example - Continued

In Recovery Routine

```
      .
      .
      .
      EAM
      LFS        0
      CØX        D (RCØVER)         Pick out proper RPW1
      ESM
      ESP        D (DECIDE)
      EAM
      LFS         1
      CØX        D (RCØVER)         Pick out proper RPW2
      ESM
      ESP        D (DECIDE)
      EAM
      LFS         2
      CØX        D (RCØVER)         Pick out proper ICW1
      ESM
      ESP        D (DECIDE)
      PCR                           Recover:  Set PSB, CSB, PC and Index
                                              registers to pre-PRR settings
PRØC .DECIDE (CØPY1, CØPY2, CØPY3) LØGICAL
DECLARE LØGICAL, CØPY1, CØPY2, CØPY3
IF CØPY1 EQ CØPY2 THEN RETURN (CØPY1)
   ELSE IF CØPY2 EQ CØPY3 THEN RETURN (CØPY2) ENDALL
EXIT (CØPY1)
```

4.13 Input/Output

The concept of viewing all input and output data as data files has been incorporated into the AGC Architecture. Initiation of input usually begins with an interrupt of the central processor. After determining the type of input to be processed,the processor executes a descriptor call (see Section 4.5.4). The descriptor call references the appropriate file descriptor in the PCT, which is interpreted and sent to the I/O control units. Once initiated, the input proceeds in an asynchronous manner to completion. The decoupling of the Input and Output from the AGC CPU functions is achieved by providing separate input and output buffer registers for both serial and parallel input and output data. Completion is indicated by setting a condition in the interrupt register. Output is usually initiated by the central processor without regard to interrupts, but this is not always the case.

The following example illustrates the steps from source SPL/MK III coding through the SPL/MK III compilation process to the initiation and execution of I/O at object time.

Example:
```
DECLARE FILE  (8 BITS) CHAN2, POSX REG6
   :
   :
INPUT POSX
```

The SPL/MK III AGC compiler will create a File Descriptor from the File Declaration and will place the File Descriptor in the PRT at address α . The File Descriptor will look like

FD(α)	00	1	1	00000010	10111	01000	0000100110
	2	1	1	8	5	5	10

The first two bits identify the descriptor as a File Descriptor (00). The next bit signifies redundant storage (1) of the item POSX into REG6, REG6+1, and REG6+2 words in scratchpad memory. The next bit (1) signifies that the file consists of 1 word records. The next 8 bits signify that when Input is requested the I/O control will utilize CHAN2. The next 5 bits signify that the item POSX begins at bit position 23 (utilizing the bit notation (0-31) of the item REG6). The next 5 bits indicate that the length of item POSX is 8 bits. The last 10 bits give the address of REG6 in scratchpad memory. Incidentally, in this example the item REG6 has been declared elsewhere in the SPL/MK III source program, as an item the length of which is 32 bits.

When the SPL/MK III compiler encounters the Input Statement "INPUT POSX", it creates a descriptor call to the FD at address α in the PRT.

At object time when the processor encounters the descriptor call in the syllable string it accesses the FD(α). The FD initiates Input by being sent to the I/O control unit. The I/O control executes the FD, which causes the item POSX of File to be input over channel 2 to REG6, REG6+1, and REG6+2 in scratchpad memory.

All file descriptors have the general format of the file descriptor (FD1) illustrated in Figure 4-17. The memory address usually represents the beginning address in scratch pad memory where the first word of the input will be placed, or where the first word of output is contained. If the input data is less than a word (a field), then the data will be contained within that word. The interpretation of the information contained in the record control portion of the file descriptor depends on the structure of the file control information.

File Descriptor 2 (FD2) is for Analog to Digital Conversion. Here the file control specifies redundant storage of the converted value in scratch pad address A, A+1, and A+2. The file is composed of a single word. The file control specifies the A/D converter to be used, and the quantity to be converted. The record control information specifies the size of the converted field (including sign) and the starting position of the field in the input/output word. The converted data will be stored in memory as a partial field of the address indicated.

File Descriptor 3 (FD3) illustrates the description of a file of one word contained in the input/output, which contains 32 input discretes. In this instance the record description is not utilized. Once in memory the settings of one or more discretes is tested by placing the word in the stack.

File Descriptor 4 (FD4) illustrates the setting of a single output discrete. Here the record control information is relevant. It specifies which one of 32 discretes to set ON or OFF. The setting of a single output discrete is a mutually exclusive operation with regard to the setting of all other discretes in the specified output discrete register. Here tne scratch pad memory address is not relevant.

File Descriptor 5 (FD5) describes a multiword output file for telemetry output. The file control description notes that there is control information contained in the top of the stack which must be sent to the Input/Output prior to transferring the file information. The control word is used to specify such things as bit rate, byte size, and formatting to the input/output for control of data transfer between the external device and the input/output unit. The record control information in this instance specifies the starting location of the output data in each word and the number of words to be transferred.

File Descriptor 6 (FD6) illustrates Digital to Analog (D/A) conversion. The file is composed of a single word. The file control specifies the D/A converter to be used, and the quantity to be converted. The record control field specifies the size of the field (including sign) to be converted, and the starting position of the field in the output word.

File Descriptor 7 (FD7) illustrates the setting of all discretes in the specified output discrete file (here a 1 word register). The OFF condition in record control indicates that the bits are to be set in the specified output device address in accordance with the bits contained in the specified scratch pad memory address.

Format for General File Descriptor

FD1	0 0	File Control	Record Control	Memory Address
	0 1	10	10	10
	Code for File Descriptor			

File Description for A/D Conversion (Input)

FD2	0 0	1	1	Device/ Channel Address	Begin Position	Length	Address
				8	5	5	10
		redundant store	single word file				

File Description for Input Discretes

FD3	0 0	0	1	Device Address			Address
				8	5	5	10
			single word file				

File Description for Setting Single Output Discrete

FD4	0 0	0	1	Device Address	1 ON 0 OFF	Discrete Number	Address
				8	5	5	10
			single word file				

File Description for Telemetry Output

FD5	0 0	1	0	Device/ Channel Address	Begin Position	Number of Words	Address
				8	5	5	10
		control word present	multiple word file				

FILE DESCRIPTORS

Figure 4-17

File Description for D/A Conversion (Output)

FD6	0 0	0	1	Device/ Channel Address	Begin Position	Length	Address
			↑ single word file		5	5	10

File Description for Setting Group of Output Discretes

FD7	0 0	0	1	Device Address	1 ON 0 OFF		Address
			↑ single word file	8	5	5	10

FILE DESCRIPTORS

Figure 4-17 (Cont'd)

5.0 Programming Studies

In the early phase of the architecture study, a set of guidance and targeting equations representative of advanced ballistic guidance and targeting functions was selected. The approved equations were developed into a guidance and targeting equation specification (reference 4). These equations were then used throughout the architecture study as the basis of conducting tradeoffs in relation to architectural configurations. The equations were used to determine the adequacy of SPL/MK II and were programmed and revised during the evolution of SPL/MK III. The equations as programmed in SPL/MK III were utilized to hand generate compiler code for various computer architecture functions under study.

This section presents the final results of the programming studies for the architecture selected. A summary of the programming studies is first presented followed by a presentation on each of the selected equations. The overall results of the programming studies are very encouraging in relation to the effective use of a higher order programming language (SPL) for efficient programming of the advanced guidance computer architecture.

The programming study for Figure 1 (reference 4) compared program size between the HDC-701P (IDCU) and the AGC architecture. This was done because the equations for Figure 1 were a modification of the model F free flight gravitation equations, block C20 of reference 1, and no direct comparison with sizing with a single address machine was possible. The programming study utilized the equations in Figure 2 (reference 4) compared the single address architecture with the AGC architecture, and covered block F11 position integration (reference 1). The programming study for the equations contained in Figure 3 (reference 4) compared the single address architecture to the AGC architecture for block T12 and T13 (reference 1). The programming study for the equations contained in Figure 4 (reference 4) compared the single address architecture with that of the AGC architecture for block T60 (reference 1). The programming study for the equations in Figure 5 (reference 4) compared the HGC-701P (IDCU) with the AGC architecture for the control equation (reference 3). The IDCU was utilized because it was felt that the sizing and timing for the D37C would not result in a representative comparison.

5.1 Programming Study Summary

The equations represented in Figures 1 and 5 have been programmed for the IDCU and for the AGC. The equations represented by Figures 2, 3 and 4 have been programmed for the AGC and compared to the instruction and word counts contained in reference 2. The Table 5-1 represents a summary of the programming study effort.

The programming effort accomplished for the equations represented by Figures 2, 3 and 4 represent the best comparison between the AGC architecture and the single address structure baseline for the ATS studies. It is best to compare the results for the equations represented by Figures 2, 3 and 4 as a group because the implementation of the equations all use library functions such as square root, arctangent and sine/cosine, and in the case of the ATS configuration, service routines such as vector cross product, and coordinate transformation. In the case of the ATS configuration, the implementation of the equations represented by Figure 2 utilized the square root routine; the implementation of the equations represented by Figure 3 utilized the arctangent, square root, vector cross product, and coordinate transformation, which in itself utilizes the sine/cosine, and matrix multiply routines; and the implementation of the equations represented by Figure 4 utilized the arctangent and the sine/cosine routines. In relation to the AGC configuration, the vector and matrix functions are performed through a combination of in line program and the vector magnitude and dot product routines. The trigometric functions are implemented as subroutines.

Summarizing the memory requirements to implement the equations for Figures 2, 3 and 4 on the AGC architecture vs the ATS configuration we have:

	AGC Memory Words	ATS Memory Words
In Line Program	76	177
Service Routines	80	193
Totals	156	370

The results of the programming studies for equations indicate that there is an overall memory reduction of 60% in the memory requirements for implementing the representative equations outlined in Figures 2, 3 and 4. This percentage appears to hold for both the in line code and for the service routines as a whole. It appears to hold for the in line programming for equations represented by Figures 2, 3 and 4 separately. There is less of an improvement noted for the trigometric functions. This is in part due to the considerations of having the service routines written in SPL/MK III. For example, the ATS sine/cosine routine is designed to be entered as either the sine or the cosine as a parameter, whereas SPL calls each separately. This requires a separate SPL procedure, which generates an extra 10 syllables or three words for the AGC configuration.

It is recognized that further analysis might reduce the difference in memory sizing between the two architectures. On the one hand one could argue that the 20% allowance for such things as shifting, scaling, and growth, for the sizing done for the ATS configuration could be significantly reduced by a careful reprogramming of the equations in Figure 2, 3 and 4. On the other hand no credit has been taken in the AGC sizing for inclusion of constants as literals in the programming string. The most significant conclusion that should be drawn from the results is that with the proper computer architecture it is feasible to utilize a higher order programming language (SPL), to generate object code which is as efficient as can be produced in assembly language for state of the art single address airborne computer architectures.

Summarizing the memory requirements to implement the equations for Figures 1 and 5 on the AGC architecture vs the IDCU we have:

	AGC Memory Words		IDCU Memory Words	
	Figure		Figure	
	1	5	1	5
In-Line Program	45	10	83	23
Service Routine	32		32	
Totals	77	10	115	23

The results of the programming study for implementing the equations for Figure 1 indicate that there is an overall memory reduction of 30%, and for Figure 5 a reduction of 55%. For both Figures 1 and 5 the comparison includes the IDCU word count for instructions and literals. The comparison for Figure 1 does not include the word count of the DIV subroutine for the IDCU as it is felt that DIV would be a required instruction to satisfy timing requirements

	ATS		IDCU		AGC		
Figure	Inst	Words	Inst	Words	Syl	Bytes	Words
1			141	83	121	180	45
2	51	26			35	58	15
3	267	134			133	212	53
4	34	17			19	32	8
5			49	23	26	40	10
ARCTAN	72	36			76	96	24
SQRT	68	34			63	76	19
SIN/COS	71	36	54	32	97	128	32
VCP	46	23					
COORD TRANS	94	47					
MATRIX MULT (3x3)	33	17					
VMAG					5	8	2
DOT					10	12	3
DIV			23	11			

Table 5-1: Programming Studies - Memory Requirements Summary

5.2 Programming Examples

The programming examples that follow represent the implementation of the equations and functions specified in reference 4. In that report each set of equations appears as a Figure (1-5), which is the method of presenting them in this report. For each example, first the figure, which comprises the equation, is presented. Next an analysis and discussion of relative considerations implementing the equation is presented. Each example is then programmed in SPL/MK III. Next the AGC code is presented based on "pseudo compilation" of the SPL code. By pseudo compilation is meant the code that would be produced by an SPL compiler generating code for the AGC architecture. In doing so the storage for local and global scalar variables, constants, arrays and descriptors, has not been explicitly represented. The selection of a 32 bit word size, closely bounds the potential differential between data representation for the ATS organization vs the AGC architecture. Finally for Figures 1 and 5 the IDCU code written in an assembly language format is presented.

5.2.1 Figure 1

INPUTS

CG1, a_e, J_{nm}, λ_{nm} - constants

R - Geocentric Radius

ϕ, λ - Latitude and Longitude

OUTPUT

g - output of test equation (no physical significance)

EQUATION

1. $$g = \frac{CG1}{R^2}\left[\sum_{n=2}^{6}\sum_{m=1}^{n} mJ_{nm}\left(\frac{a_e}{R}\right)^n \frac{P_n^m}{\cos\phi}\sin m(\lambda - \lambda_{nm})\right]$$

(Reference 1 Page 7-303)

Note: $P_n^m = 1$ for $m = 0$ or $n = 1$

$P_n^m = P_n^{m-1}\sin\phi + P_{n-1}^m\cos\phi$ for $m \geq 1$ and $n \geq 2$

(Reference 1 Page 7-303)
(Modified)

Figure 1

SCALING

Quantity	Maximum Bit Required	Least Bit Required
ϕ	+6	-13
λ	+8	-13
R	+1	-22
a_e	+1	-22
CG1	+1	-22
λ_{nm}	+6	-13
J_{nm}	+1	-22
P_n^m	+1	-22
g	+0	-22

TIMING

Maximum execution time is 6000 μ sec

Figure 1 (contd)

5.2.1.1 Figure 1 Analysis

Figure 1 has been programmed as a procedure which computes the value of

$$g(\phi,\lambda) = \frac{CG1}{R^2 \cos\phi} \sum_{n=2}^{6} \left(\frac{a_e}{R}\right)^n \sum_{m=1}^{n} m\, J_{nm}\, P_n^m \sin(m(\lambda-\lambda_{nm}))$$

where J_{nm}, λ_{nm} are constant arrays
a_e, R, CG1 are constants
globally defined

and $P_n^m = P_m^{n-1} \cos\phi + P_{m-1}^n \sin\phi$; $n > 1$, $m > 0$

$$P_m^1 = P_0^n \equiv 1$$

The variables ϕ, λ and constants λ_{nm} are angles in degrees which are converted to radians by multiplying by the constant D2R.

The arrays J_{nm}, λ_{nm} are stored in one-dimensional arrays JMN and LMN respectively. This results in a saving of 10 words for storing each array (20 x 1 versus 5 x 6). The index K used to subscript these arrays is initialized to 0 before the computation begins and is incremented in the innermost loop (on m). This avoids using a complicated formula in m and n.

The quantity $(a_e/R)^n$ is represented in the program as NPOW. This is initialized to AER = a_e/R and updated by multiplying it by AER each time it is used to modify the inner sum.

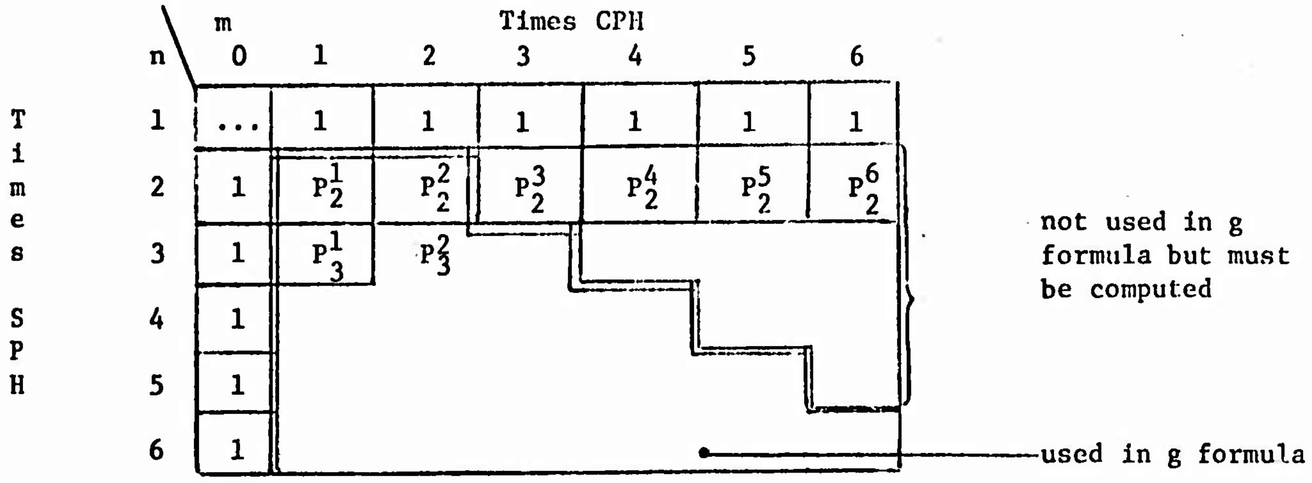
The quantities sin ϕ and cos ϕ are computed only once before the main computation starts (SPH, CPH respectively).

The quantities P_n^m could have been computed recursively by writing

```
PROC .P(M,N) F 7  DECLARE I, M, N
IF M EQ 0 OR N EQ 1 THEN RETURN(1) ELSE
RETURN (CPH*.P(M,N-1) + SPH*.P(M-1,N)) END
EXIT
```

This would have resulted in a very short code for P_n^m but would have resulted in excessive execution time and stack depth (4 words added to stack for each level of call on .P).

To avoid recursion, the computation for P_n^m can be visualized as the following table. Except for the bordering values of 1, each quantity is computed by adding together the products left-neighbor times SPH and above neighbor times CPH:



n \ m	0	1	2	3	4	5	6
1	...	1	1	1	1	1	1
2	1	P_2^1	P_2^2	P_2^3	P_2^4	P_2^5	P_2^6
3	1	P_3^1	P_3^2				
4	1						
5	1						
6	1						

Example: $P_3^2 = P_2^2 * \text{CPH} + P_3^1 * \text{SPH}$

Instead of retaining this whole 6 x 7 table we need only retain the 6-element row of above neighbors A (initialized to all ones) and the 7-element row of left neighbors (with the first element always = 1). The gyst of the computation then is: every time m=1, replace the A-row by the old L-row and compute a new L-row by the formula

$$L_i = A_{i-1} * \text{CPH} + L_{i-1} * \text{SPH}$$

The computation for P_n^m as written requires two tests each time entered. These are 1) does m=1 and n-2 and if not 2) does m=1. These could be eliminated by using a 20-way switched GØTØ but the 20 words plus descriptor required for the transfer vector were deemed prohibitive.

The coding for Figure 1 as per the foregoing analysis results in the following ins.ruction and byte counts (control stream only):

	No. Syllables	No. Bytes
.G	71	112 (2 waste)
.PMN	50	68 (1 waste)
Total	121	180 (3 waste)

The maximum stack depth is 19 words if one assumes that the stack usage of neither .SIN or .COS exceeds the 6 words used by .PMN. The stack depth is reckoned as follows:

<u>Call .G from outside</u>

2 RPW's	2
Phi	1
Lamda	1
<u>Depth in .G when .PMN is called</u>	7
<u>Call .PMN : 2 RPW's</u>	2
<u>Maximum .PMN depth</u>	6
	19

PROGRAM SPL/MK III Code for Figure I of WS-1008-7-2

PROGRAMMER S.J. Nersan DATE 23 Sept. 1970

```
PRØC .G(PHI,LAMDA)  F7  ''CGI,AE,R,JMN,LMN ARE GLØBAL''
      DECLARE F7,  SUM, GUM, SPH, CPH, NPØW, AER
      DECLARE I   ,  I   ,  M   ,  N   , K

      GUM  = 0
      NPØW  = AER  = AE/R
      CPH   = D2R*PHI  ''D2R=DEG TØ RAD, GLØBAL CØNSTANT''
      SPH   = .SIN(CPH)
      CPH   = .CØS(CPH)
      K     = 0

      FØR  N=2 TØ 6    SUM  = 0

           FØR  M=1  TØ  N
           SUM  = SUM+M*JMN(K)*.PMN*.SIN(D2R*M*(LAMDA-LMN(K)))
           K    = K + 1
           END

      GUM  = GUM + SUM * NPØW = NPØW * AER
      END
```

PROGRAM

PROGRAMMER

DATE

```
 PRØC .PMN F 7 ''THIS PRØC IS NESTED INSIDE PRØC .G''
        DECLARE (6) F 7, A,LL,ØNES=(6(1)),L(7),LO ITEM = 1
        ØVERLAY LO=L(0), LL(0)=L(1)

        IF N EQ 2 AND M EQ 1 THEN A=ØNES GØTØ PCØM END

        IF M EQ 1 THEN               A = LL
                       PCØM. FØR I=1 TØ 6
                             L(I) = L(I-1)*SPH + A(I-1)*CPH
                             ENDALL

        EXIT(L(M))                    ''VALUE LINE ØF PMN''

      EXIT(GUM*CG1/(R*R*CPH))       ''VALUE LINE ØF G''
```

PROGRAM AGC CODE : Figure 1 WS-1008-7-2

PROGRAMMER

DATE

Stack Depth	BYTE	LABEL	OPERATION	ADDRESS	COMMENT
0	0	.G	SSZ	GUM	GUM=0
1	2		FST	AE	
2	4		FST	R	
1	6		÷RS		NPØW=AER=AE/R
1	7		SRK	AER	RESULT IS RØUNDED
0	9		STC	NPØW	
1	11		FST	D2R	
2	13		FSC	3	D2R-GLOBAL CØNSTANT
1	15		*S		CPH=D2R*PHI PHI-CALL SEQ ARGUMENT
1	16		SRK	CPH	
1	18		ESP	D(SIN)	SPH=.SIN(CPH)
0	20		STC	SPH	
1	22		FSS	CPH	
1	24		ESP	D(CØS)	CPH=.CØS(CPH)
0	26		STC	CPH	
0	28		SSZ	K	K=0
1	30		LFS	AD(N)	
2	32		LFS	1	SET UP FØR-LØØP ØN N
3	33		LFS	6	AD(N) IS SPM ADDRESS ØF N
4	34		LFS	1	
2	35		XCR		
2	36	LRPN	SSZ	SUM	SUM=0; THIS IS LRP ØF N-LØØP

PROGRAM

PROGRAMMER DATE

```
3   38       LFS     AD(M)
4   40       LFS     1
5   41       FCV                   SET UP FØR-LØØP ØN M
6   42       LFS     1                 FCV FETCHES N
4   43       XCR
5   44  LRPM FSS     SUM
6   46       FCV
7   47       FSS     K
7   49       ELF     D(JMN)
6   51        *S
7   52       ESP     D(PMN)
6   54        *S
7   55       FST     D2R
8   57       FCV
7   58        *S                    SUM= SUM +
8   59       FSC     2
9   61       FSS     K                   M*JMN(K)*.PMN*
9   63       ELF     D(LMN)
8   65        -S                           .SIN(D2R*M*(LAMDA-LMN(K)))
7   66        *S
7   67       ESP     D(SIN)         FCV FETCHES M
6   69        *S
5   70        +S
4   71       SRC     SUM
```

PROGRAM /

PROGRAMMER   DATE

```
5     73   FSS      K
6     75   LFS      1              K=K+1
5     76    +S
4     77   STC      K
4/2   79   NDX                 LØØP TØ LRPM
3     80   FSS      NPØW
4     82   FSS      AER
3     84    *S
3     85   SRK      NPØW           GUM= GUM+SUM*NPØW=NPØW*AER
4     87   FSS      SUM
3     89    *S
4     90   FSS      GUM
3     92    +S
2     93   SRC      GUM
2/0   95   NDX                 LØØP TØ LRPN
1     96   FSS      GUM
2     98   FST      CGI
1    100    *S
2    101   FST      R              CGI*GUM/(R*R*CPH) TØ TØS AS
3    103   DUP
2    104    *S                     FUNCTION RESULT
3    105   FSS      CPH
2    107    *S
1    108    ÷RS
```

PROGRAM

PROGRAMMER   DATE

```
1 1.09      NSX           RETURN FROM PROC.G
   110      NOP          }WASTE (TO FILL OUT LAST WORD)
   111      NOP          }
```

PROGRAM AGC Code: Figure 1 WS-1008-7-2

PROGRAMMER

DATE

Stack Depth	BYTE	LABEL	OPERATION	ADDRESS	COMMENT
1	0	.PMN	FSS	N	N EQ 2 AND M EQ 1
2	2		LFS	2	
1	3		EQ		
2	4		FSS	M	
3	6		LFS	1	
2	7		EQ		
1	8		AND		
2	9		LFS	AD(F1)	LØAD ADDRESS ØF FALSE BRANCH
2/0	10		JØF		JUMP ØN FALSE
0	11		EAM		A=ØNES ARRAY ASSIGNMENT
6	12		AFC	D(ØNES)	
6	14		SSO		
0	15		ASC	D(A)	
0	17		ESM		
1	18		LFS	AD(PCØM)	GØTØ PCØM
1/0	20		JMP		
1	21	F1	FSS	M	M EQ 1
2	23		LFS	1	
1	24		EQ		
2	25		LFS	AD(F2)	LØAD ADDRESS ØF FALSE BRANCH
2/0	27		JØF		JUMP ØN FALSE

PROGRAM

PROGRAMMER

DATE

```
0   28          EAM                 ⎫
6   29          AFC       D(LL)     ⎪
6   31          SSO                 ⎬  A=LL ARRAY ASSIGNMENT
0   32          ASC       D(A)      ⎪
0   34          ESM                 ⎭
1   35   PCΦM.  LFS       AD(I)     ⎫
2   37          LFS       1         ⎪
3   38          LFS       6         ⎬  SET UP FΦR-LΦΦP ΦN I
4   39          LFS       1         ⎪
2   40          XCR                 ⎭
3   41   LRPI   FCV                 ⎫
4   42          LFS       1         ⎪
3   43          -S                  ⎪
3   44          ELF       D(L)      ⎪
4   46          FSS       SPH       ⎪
3   48          *S                  ⎪
4   49          FCV                 ⎬  TΦS=SPH*L(I-1)+CPH*A(I-1)
5   50          LFS       1         ⎪
4   51          -S                  ⎪
4   52          ELF       D(A)      ⎪
5   54          FSS       CPH       ⎪
4   56          *S                  ⎪
3   57          +S                  ⎭
```

PROGRAM

PROGRAMMER

DATE

```
   58     FCV              } L(I) = TØS
   59     ESC       D(L)   }
   61     NDX              LØØP TØ LRPI
   62     FSS       M      } L(M) TØ TØS AS FUNCTIØN RESULT
   64     ELF       D(L)   }
   66     NSX              RETURN FRØM PRØC .PMN
   67     NØP              WASTE (TØ FILL ØUT LAST WØRD)
```

5.2.2 **Figure 2**

INPUTS:

Δt_{nav} - navigation step size

$\dot{X}$ - old velocity component

X_{ℓ} , X - old position component as carried in double precision. X and X_{ℓ} denote the most and the least significant portion of the variable, respectively.

$\ddot{X}_g$ - old gravity component

CMP - a mask/program constant

$\Delta\dot{X}_s$ - increment of non-gravitational acceleration

OUTPUTS:

X - new position component

X_{ℓ} - new position, least significant part

$(\ddot{X}_g)_{-1}$ - updated gravity component

Figure 2

EQUATIONS:

2. $\Delta X = [\dot{X} + (\Delta\dot{X}_g + \ddot{X}_g\,\Delta t_{nav})(1/2)]\,\Delta t_{nav}$

(Reference 1
Page 7-84)

3. $(\ddot{X}_g)_{-1} \leftarrow \ddot{X}_g$

(Reference 1
Page 7-84)

4. $\Delta X' = \Delta X + X_\ell$

(Reference 1
Page 7-84)

5. $X_\ell \leftarrow \Delta X' \wedge (CMP)$

(Reference 1
Page 7-84)

6. $X \leftarrow X + \Delta X'$

(Reference 1
Page 7-84)

Figure 2 (contd)

SCALING

Quantity	Maximum Bit Required	Least Bit Required
ΔX	-7	-25
$\overline{X}$	-1	-26
$\Delta \dot{X}_g$	-6	-26
$\ddot{X}_g$	0	-20
Δt_{nav}	-8	-28
X, X_ℓ	+1	-25
$\Delta X'$	$\geq$ -7	-25

TIMING

Maximum execution time is 170 μsec

Figure 2 (contd)

5.2.2.1 Figure 2 Analysis

Figure 2 as presented in WS-1008-7-2 is incomplete. The equations presented there deal only with the X-component of the position vector computation. The computation must be repeated for the Y and Z-components. Finally, the magnitude of the position vector $\sqrt{X^2 + Y^2 + Z^2}$ must be computed.

On the AGC, these computations are best handled as whole vector calculations. Accordingly we define

$R = (X, Y, Z)$

$DR = (\Delta X, \Delta Y, \Delta Z)$

$RD\O T = (\dot{X}, \dot{Y}, \dot{Z})$

$RGDD = (\ddot{X}_g, \ddot{Y}_g, \ddot{Z}_g) \; || \; RG\O LD = RGDD_{-1}$ (old values)

$DRSD = (\Delta\dot{X}_s, \Delta\dot{Y}_s, \Delta\dot{Z}_s)$

$DTNAV = \Delta t_{nav}$

The vector equations for couplet operation are

$$\left.\begin{array}{ll}
\left.\begin{array}{l}
\text{DR = DTNAV * RGDD}\\
\text{DR = DRSD + DR}\\
\text{DR = DR/2}\\
\text{DR = RDØT + DR}\\
\text{DR = DTNAV * DR}
\end{array}\right\} & \text{(Equation 2)}\\
\text{R = R + DR} & \text{(Equation 6)}\\
\text{RGOLD = RGDD} & \text{(Equation 3)}
\end{array}\right\}\ \text{Array Mode}$$

$$\text{DIST = .SQRT (R' * R)} \qquad \Big|\,\text{Scalar Mode}$$

Equations 4 and 5 of Figure 2 are fixed point artifacts which do not need to be considered on a floating point computer. Each of the vectors is represented by a descriptor which contains the information that those vectors which need to be are stored in double precision. Array mode operation will automatically provide double precision operations when the operands are double precision.

The statistics for Figure 2 are

	No. Syllables	No. Bytes
FIG 2	35	58

The maximum stack depth is likely to be

$$3 + \text{MSD (SQRT)} > 6$$

because MSD (SQRT) would probably > 3. The next highest stack depth is 6 when both stacks 0 and 1 are holding vectors prior to an operation.

PROGRAM

PROGRAMMER

DATE

BYTE	LABEL	OPERATION	ADDRESS	COMMENT	Depth of Stack: 0	1	2
0		EAM		ENTER ARRAY MΦDE	0	0	0
1		FSS	DTNAV		1	0	0
3		AFC	D(RGDD)	DR = DTNAV * RGDD	1	3	0
5		AE*			0	0	3
6		ASC	D(DR)		0	0	0
8		AFC	D(DRSD)		3	0	0
10		AFC	D(DR)	DR = DRSD + DR	3	3	0
12		AE+			0	0	3
13		ASC	D(DR)		0	0	0
15		AFC	D(DR)		3	0	0
17		LFS	2	DR = DR / 2	3	1	0
18		AE÷			0	0	3
19		ASC	D(DR)		0	0	0
21		AFC	D(RDΦT)		3	0	0
23		AFC	D(DR)	DR = RDΦT + DR	3	3	0
25		AE+			0	0	3
26		ASC	D(DR)		0	0	0
28		FSS	DTNAV		1	0	0
30		AFC	D(DR)	DR = DTNAV * DR	1	3	0
32		AE*			0	0	3
33		ASC	D(DR)		0	0	0

PROGRAM

PROGRAMMER DATE

				Depth of Stack: 0	1	2
35	AFC	D(R')		3	0	0
37	AFC	D(DR)	R = R + DR	3	3	0
39	AE+			0	0	3
40	ASC	D(R)		0	0	0
42	AFC	D(RGDD)		3	0	0
44	SSD		RGØLD = RGDD	3	0	0
45	ASC	D(RGØLD)		0	0	0
47	AFR	D(R)		3	0	0
49	AFC	D(R)	TØS = R'*R	3	3	0
51	MXM			0	0	1
52	FSR			1	0	0
53	ESM		ENTER SCALAR MODE	1	–	–
54	ESP	D(SQRT)	DIST = .SQRT(TØS)	1	–	–
56	STC	DIST		0	–	–

5.2.3 Figure 3

INPUTS

Ω_e = earth's rotational rate

$\bar{R}_L$, R_L - launch position vector, F frame, and its magnitude

$\bar{R}'_A$ = aim point, F frame

b_e - earth's polar radius

t_{Ir} = reference impact time

Z_L = Z component of launch position vector

OUTPUTS

ϕ = range angle between $\bar{R}_L$ and $\bar{R}_A$

A_f = firing azimuth estimate for great circle defined by $\bar{R}_L$ and $\bar{R}_A$

EQUATIONS

7. $\bar{R}_A = \left[3, -\Omega_e t_{Ir}\right]\bar{R}'_A$ (Reference 1 Page 7-225)

8. $\bar{P}_{11} = \bar{R}_L \times \bar{R}'_A$ (Reference 1 Page 7-225)

9. $P_{11} = (\bar{P}_{11} \cdot \bar{P}_{12})^{\frac{1}{2}}$ (Reference 1 Page 7-225)

10. $P_{12} = \bar{R}_L \cdot \bar{R}_A$ (Reference 1 Page 7-225)

Figure 3

EQUATIONS (continued)

11. $\Phi = \tan^{-1}\left(\frac{P_{11}}{P_{12}}\right); \; 0 < \Phi < \pi$

(Reference 1
Page 7-226)

12. $\bar{P}_1 = \bar{R}_A - \frac{P_{12}}{R_L^2}\bar{R}_L$

(Reference 1
Page 7-227)

13. $\hat{P}_1 = \frac{\bar{P}_1}{(\bar{P}_1 \cdot \bar{P}_1)^{1/2}}$

(Reference 1
Page 7-227)

14. $\bar{P}_2 = \begin{bmatrix} 0 \\ 0 \\ b_e \end{bmatrix} - Z_L \frac{\bar{R}_L}{R_L}$

(Reference 1
Page 7-228)

15. $\hat{P}_2 = \frac{\bar{P}_2}{(\bar{P}_2 \cdot \bar{P}_2)^{1/2}}$

(Reference 1
Page 7-228)

16. $\bar{P}_3 = \hat{P}_1 \times \hat{P}_2$

(Reference 1
Page 7-228)

17. $P_4 = \mathrm{sgn}\,(\bar{P}_3 \cdot \bar{R}_L)$

(Reference 1
Page 7-228)

18. $P_5 = P_4(\bar{P}_3 \cdot \bar{P}_3)^{1/2}$

(Reference 1
Page 7-228)

19. $P_6 = \hat{P}_1 \cdot \hat{P}_2$

(Reference 1
Page 7-228)

20. $A_f \overset{\sim}{=} \tan^{-1}\left(\frac{P_5}{P_6}\right); \; -\pi < A_f \leq \pi$

(Reference 1
Page 7-228)

Figure 3 (Part 2)

SCALING

Quantity	Maximum Bit Required	Least Bit Required
Ω_e	-8	-25
$\bar{R}_L$	+1	-22
R_L	+1	-22
$\bar{R}'_A$	+1	-22
b_e	+1	-22
t_{Ir}	+11	-7
ϕ	+7	-13
A_f	+7	-7
$\bar{R}_A$	+1	-22
$\bar{P}_{11}$	+1	-22
P_{11}	+1	-22
P_{12}	+1	-22
$\bar{P}_1$	+1	-18
$\hat{P}_1$	+1	-18
$\bar{P}_2$	+1	-18
Z_L	+1	-22

Figure 3 (contd)

SCALING (contd)

Quantity	Maximum Bit Required	Least Bit Required
$\hat{P}_2$	+1	-18
$\bar{P}_3$	+1	-18
P_4	+1	+1
P_5	+1	-18
P_6	+1	-18

TIMING

Maximum execution time is 5800 μsec

Figure 3 (contd)

5.2.3.1 Figure 3 Analysis

The only things that require comment regarding the computations of Figure 3 are that since many of the equations are array or vector formulas, one must take care to break the formula up for array mode couplet operation. Thus, for example, equation 14

$$\bar{P}_2 = \underset{\text{PVEC}}{\begin{bmatrix} 0 \\ 0 \\ b_e \end{bmatrix}} - \underset{\substack{\big\uparrow \\ \bar{R}_L(3)}}{Z_L} \frac{\bar{R}_L}{R_L}$$

becomes

```
T1 = RBL(3)/RL
PB2 = T1   *RBL
PB2 = PVEC-PB2
```

A second item requiring comment is that we assume that the system library .ARCTAN routine will provide a result in the range $-\pi/2 \leq \text{result} \leq \pi/2$, hence, equations 11 and 20 require additional analysis:

$$\text{Equation 11:} \quad \Phi = \tan^{-1}\left(\frac{P_{11}}{P_{12}}\right) \; ; \; 0 < \Phi < \pi$$

The additional analysis is

$$\text{IF } \Phi < 0,\ \Phi = \Phi + \pi$$

This is not most conveniently written (from the viewpoint of object code compactness)

```
PHI = (PHI LS 0) * PI + PHI = .ARCTAN (etc.)
```

Equation 20:

$$A_f = \tan^{-1}\left(\frac{P_5}{P_6}\right);\ -\pi < A_f < \pi$$

The assumption is made that the quandrant is to be selected on the basis of considering P_5 to be the X-coordinate and P_6 the Y-coordinate. Hence,

If P_5 and P_6 are both < 0, $A_f = A_f - \pi$

If $P_5 < 0$ but $P_6 \geq 0$, $A_f = A_f + \pi$

If $P_5 \geq 0$, A_f = As is

This is most compactly written

```
AF = .ARCTAN (P5/P6) + PI * (P5 LS 0) * .SIGN (P6)
```

Finally, it should be noted that all of the dot products have been moved into a .DOT procedure as have all the vector magnitudes (.VMAG). This permits scalar formulas to be written without consideration of array mode couplet requirements.

The statistics for Figure 3 are:

	No. Syllables	No. Bytes	
.FIG3	133	212	(3 waste)
.VMAG	5	8	(0 waste)
.DOT	10	12	(3 waste)
Total	148	232	(6 waste)

The stack depth shown in the coding pertains to stack 0. The maximum stack depth is 9 words encountered during the rotation array multiplication.

PROGRAM SPL Code: Figure 3 WS-1008-7-2

PROGRAMMER

DATE

```
PRØC .FIG3(=PHI,AF)

 ''SCALARS''
      DECLARE F 7,  T1,P12,P5,P6
      ØVERLAY       T1 = P12=P5

 ''3 BY 1 VECTØRS''
      DECLARE (3,1) F 7, RBA,PH1,PB11,PB1,PB2,PB3,PH2
      ØVERLAY            RBA=PH1,PB11=PB1=PB2=PB3
      DECLARE,PVEC (3,1) F 7 C =(0,0,BE) ''SUBSTITUTE ACTUAL
                                          VALUE FØR BE PRIØR
                                          TØ CØMPILATIØN''

 ''3 BY 3 RØTATIØN MATRIX''
      DECLARE, RØTATE (3,3) F 7 =(8(0),1)

 ''GLØBALLY DEFINED...ØMEGAE,TIR,RØPA,RBL,RL''

 ''ANALYSIS ASSUMES -PI/2 LQ ARCTAN LQ PI/2''
```

PROGRAM SPL Code: Figure 3 WS-100B-1-2

PROGRAMMER

DATE

ARRAY MODE STATEMENTS

SCALAR MODE STATEMENTS

```
 T1       = ΦMEGAE * TIR
 RΦTATE(0,0) = -RΦTATE(1,1) = .SIN(T1)
 RΦTATE(0,1) = RΦTATE(1,0) = .CΦS(T1)
RBA       = RΦTATE * RCPA
PB11      = RBL /* RBA
 P12      = .DΦT(RBL,RBA)
 PHI      = (PHI LS 0)*PI + PHI : .ARCTAN(.VMAG(PB11)/P12)
 T1       = P12 / RL**2
PB1       = T1 * RBL
PB1       = RBA - PB1
 T1       = .VMAG(PB1)
PH1       = PB1 / T1
 T1       = RBL(3) / RL
PB2       = T1 * RBL
PB2       = PVEC - PB2
 T1       = .VMAG(PB2)
PH2       = PB2 / T1
PB3       = PH1 /* PH2
 P5       = .SIGN(.DΦT(PB3,RBL))*.VMAG(PB3)
 Φ6       = .DΦT(PH1,PH2)
 AF       = .ARCTAN(P5/P6)+PI*(P5 LS 0)*.SIGN(P6)

EXIT
```

PROGRAM SPL CODE : FIGURE 3 WS-1008-7-2

PROGRAMMER

DATE

```
PRΦC .VMAG(V) F 7
EXIT(.SQRT(.DΦT(V,V)))

PRΦC .DΦT(V,W) F 7
      DECLARE (W) F 7, V,W ''DUMMY DIMENSIΦNS''
      EXIT(V'*W)
```

PROGRAM AGC Code: Figure 3 WS-1008-7-2

PROGRAMMER

DATE

stack depth	BYTE	LABEL	OPERATION	ADDRESS	COMMENT
1	0	FIG3.	FST	ΦMEGAE	
2	2		FSS	TIR	T1 = ΦMEGAE*TIR
1	4		*S		
1	5		SRK	T1	
1	7		ESP	D(SIN)	.SIN(T1) TΦ TΦS
2	9		LFS	1	
3	10		DUP		
1	11		ESK	D(RΦTATE)	RΦTATE(1,1) = .SIN(T1)
1	13		CHS		-.SIN(T1) TΦ TΦS
2	14		LFS	0	
3	15		DUP		
0	16		ESC	D(RΦTATE)	RΦTATE(0,0) = -.SIN(T1)
1	18		FSS	T1	
1	20		ESP	D(CΦS)	
2	22		LFS	1	RΦTATE(1,0) = .CΦS(T1)
3	23		LFS	0	
1	24		ESK	D(RΦTATE)	
2	26		LFS	0	
3	27		LFS	1	RΦTATE(0,1) = .CΦS(T1)
3	25		ESC	D(RΦTATE)	

PROGRAM AGC Code: Figure 3 WS-1008-1-2

PROGRAMMER

DATE

```
0  30      EAM
9  31      AFR     D(RØTATE)
9  33      AFC     D(RBPA)
0  35      MXM                    RBA=RØTATE*RBPA
0  36      ASC     D(RBA)            MATRIX MULTIPLY
6  38      XPF     D(RBL)
6  40      XPF     D(RBA)         PB11=RBL /* RBA
0  42      VXP                       VECTØR CRØSS PRODUCT
0  43      ASC     D(PB11)
0  45      ESM
1  46      FST     D(RBL)R
2  48      FST     D(RBA)
1  50      ESP     D(DØT)
0  52      STC     P12
1  54      FST     D(PB11)        PHI=.ARCTAN(.VMAG(PB11)/P12)
1  56      ESP     D(VMAG)
2  58      FSS     P12
1  60      ÷RS
1  61      ESP     D(ARCTAN)
2  63      FSC     3
1  65      ←K
2  66      DUP
3  61      LFS     0
2  68      LS
```

PROGRAM A&C Code: Figure 2 WS-100E-7-21

PROGRAMMER

DATE

3	69	FST	PI	
2	71	*S		PHI = (PHI LS 0)*PI + PHI
1	72	+S		
2	73	FSC	3	
0	75	←C		
1	76	FSS	P12	
2	78	FSS	RL	
3	80	DUP		T1 = P12/RL**2
2	81	*S		
1	82	÷RS		
0	83	SRC	T1	
0	85	EAM		
1	86	FSS	T1	
1	88	AFC	D(RBL)	PB1 = T1 *RBL
0	90	AE*		
0	91	ASC	D(PB1)	
3	93	AFC	D(RBA)	
3	95	AFC	D(PB1)	PB1 = RBA-PB1
0	97	AE-		
0	98	ASC	D(PB1)	
0	100	ESM		
1	101	FST	D(PB1)	T1 = .VMAG(PB1)
1	103	ESP	D(VMAG)	
0	105	STC	T1	

PROGRAM AGC Code: Figure 3 WS-1008-7-2

PROGRAMMER

DATE

```
0  107     EAM                    }
3  108     AFC     D(PB1)         }
3  110     FSS     T1             }
0  112     AE÷                    }  PHI = PB1 / T1
0  113     ASC     D(PHI)         }
0  115     ESM                    }
1  116     LFS     3              }
1  119     ELF     D(RBL)         }
2  121     FSS     RL             }  T1 = RBL(3) / RL
1  123     ÷RS                    }
0  124     SRC     T1             }
0  126     EAM                    }
1  127     FSS     T1             }
1  129     AFC     D(RBL)         }  PB2= T1 * RBL
0  131     AE*                    }
0  132     ASC     D(PB2)         }
3  134     AFC     D(PVEC)        }
3  136     AFC     D(PB2)         }
0  138     AE-                    }  PB2 = PVEC - PB2
0  139     ASC     D(PB2)         }
0  141     ESM                    }
1  142     FST     D(PB2)         }
1  144     ESP     D(VMAG)        }  T1 = .VMAG(PB2)
0  146     STC     T1             }
```

PROGRAM AGC CODE: FIGURE 3 WS-1008-7-2

PROGRAMMER

DATE

```
0  148   EAM                  }
0  149   AFC   D(PB2)         }
2  151   FSS   T1             }  PH2 = PB2 / T1
0  152   AE÷                  }
0  154   ASC   D(PH2)         }
6  156   XPF   D(PH1)         }
6  158   XPF   D(PH2)         }
0  160   VXP                  }  PB3 = PH1 /* PH2
0  161   ASC   D(PB3)         }
0  162   ESM                  }
1  164   FST   D(PB3)         }
2  166   FST   D(RBL)         }
1  168   ESP   D(DPT)         }
1  170   SGN                  }  P5 = .SIGN(.DPT(PB3, RBL)) * .VMAG(PB3)
2  171   FST   D(PB3)         }
2  173   ESP   D(VMAG)        }
1  175   *s                   }
0  176   SRC   P5             }
1  177   FST   D(PH1)         }
2  180   FST   D(PH2)         }  P6 = .DPT(PH1, PH2)
1  182   ESP   D(DPT)         }
0  184   STC   P6             }
```

PROGRAM AGC Code: Figure 3 WS-1008-7-2

PROGRAMMER

DATE

```
1  186    FSS   P5      }
2  188    FSS   P6      }
1  190     ÷RS          }
1  191    ESP   D(ARCTAN)
2  193    FST   PI      }
3  195    FSS   P5      }
4  197    LFS   0       }  AF=.ARCTAN(P5/P6)
3  198     LS           }
2  199     *S           }     +PI*(P5 LS 0)*.SIGN(P6)
3  200    FSS   P6      }
3  202    SGN           }
2  203     *S           }
1  204     +S           }
2  205    FSC   2       }
0  207    ←C            }
0  208    NSX            RETURN FROM PROC .FIG3
   209    NOP           }
   210    NOP           } WORD FILLER
   211    NOP           }
```

PROGRAM AGC Code: Figure 3 WS-1008-7-2

PROGRAMMER

DATE

Stack Depth	BYTE	LABEL	OPERATION	ADDRESS	COMMENT
1	0	.VMAG	FSC	2	PICK UP ARGUMENT
2	2		DUP		TWICE
1	3		ESP	D(DØT)	DØT PRØDUCT
1	5		ESP	D(SQRT)	SQUARE RØØT
1	7		NSX		RETURN FRØM PRØC .VMAG
0	0	.DØT	EAM		ENTER ARRAY MØDE
N	1		AFC, C	3	DESCRIPTØR IS IN CALL SEQ
N	3		AFC, C	2	DITTØ
0	5		MXM		MATRIX MULTIPLY
1	6		FSR		SCALAR RESULT TØ TØS
1	7		ESM		ENTER SCALAR MØDE
1	8		NSX		RETURN FRØM PRØC .DØT
	9		NØP		WØRD FILLER
	10		NØP		.
	11		NØP		V

5.2.4 Figure 4

INPUTS:

$\frac{\partial M_d}{\partial \dot{X}}$, $\frac{\partial M_d}{\partial \dot{Z}}$, $\frac{\partial M_c}{\partial \dot{X}}$, $\frac{\partial M_c}{\partial \dot{Y}}$, $\frac{\partial M_c}{\partial \dot{Z}}$ - miss coefficients

OUTPUTS:

θ_{fdv} - final desired vernier pitch attitude

θ_{fd3} - final desired Stage III pitch attitude

ϕ_{fdv} - final desired veriner roll attitude

EQUATIONS:

21. $\theta_{fdv} = \tan^{-1}\left[\frac{\partial M_d/\partial \dot{Z}}{\partial M_d/\partial \dot{X}}\right]; -\frac{\pi}{2} < \theta_{fdv} < \frac{\pi}{2}$ (Reference 1 Page 7-250)

22. $\theta_{fd3} = \theta_{fdv}$ (Reference 1 Page 7-250)

23. $\phi_{fdv} = \tan^{-1}\left[\frac{\cos\theta_{fdv}\,(\partial M_c/\partial \dot{Z}) - \sin\theta_{fdv}\,(\partial M_c/\partial \dot{X})}{\partial M_c/\partial \dot{Y}}\right]; -\frac{\pi}{2} < \phi_{fdv} < \frac{\pi}{2}$

(Reference 1 Page 7-250)

Figure 4

SCALING

Quantity	Maximum Bit Required	Least Bit Required
$\frac{\partial M_d}{\partial \dot{X}}$	+1	-15
$\frac{\partial M_d}{\partial \dot{Z}}$	+1	-15
$\frac{\partial M_c}{\partial \dot{X}}$	+1	-15
$\frac{\partial M_c}{\partial \dot{Y}}$	+1	-15
$\frac{\partial M_c}{\partial \dot{Z}}$	+1	-15
θ_{fdv}	+6	-5
θ_{fd3}	+6	-5
ϕ_{fdv}	+6	-5

TIMING

Maximum execution time is 1300 μsec

Figure 4

5.2.4.1 Figure 4 Analysis

The computation is very straightforward. The programmer has rearranged the order of the computation to take advantage of assumed properties of the AGC compiler. Namely, it is assumed that each time a statement is begun a check is made to see if the last thing stored in the previous statement is the first thing fetched for the new statement. If the new statement does not have a label, is not a loop return point or otherwise is not the target of a jump (e.g., ELSE clause of an IF statement) then when such a match is found the store can be changed to store and keep and the fetch can be dropped.

Thus writing

```
THETV = THET3= etc.  (not THET3 = THETV = etc.)
PHIV = .ARCTAN ((.COS (THETV) * MCZ - etc.))
                  (not MCZ * .COS(THETV))
```

guarantees that the interface of the two statements will be

```
                Condensed
STC THETV      ----------> STK THETV

FSS THETV
```

The statistics for Figure 4 are

	No. Syllables	No. Bytes	
.FIG4	19	32	(0 waste)

The maximum stack depth is

3 + Max Stack Depth of .ARCTAN

Figure 4 has been coded on the assumption that everything, input and output, is globally defined.

PROGRAM SPL CODE : FIGURE 4 WS-1008-7-2

PROGRAMMER

DATE

```
PROC .FIG4 ''EMPTY CALLING SEQUENCE''

      THETV = THET3 = .ARCTAN(MOZ/MOX)

      PHIV  = .ARCTAN((.COS(THETV)*MCZ-.SIN(THETV)*MCX)/MCY)

EXIT
```

PROGRAM AGC Code : Figure 4 WS- 1008-7-2

PROGRAMMER

DATE

```
stack Depth
↓  BYTE  LABEL    OPERATION  ADDRESS      COMMENT
1     0  FIG4.    FSS        MDZ
2     2           FSS        MDX
1     4           ÷RS                     THETV=THET3=.ARCTAN (MDZ/MDX)
1     5           ESP        D(ARCTAN)
1     7           STK        THET3
1     9           STK        THETV
1    11           ESP        D(COS)
2    13           FSS        MCZ
1    15           *S
2    16           FSS        THETV
2    18           ESP        D(SIN)
3    20           FSS        MCX          PHIV=.ARCTAN(
2    22           *S
1    23           -S                       (.COS(THETV)*MCZ-.SIN(THETV)
2    24           FSS        MCY
1    26           ÷RS                          *MCX)/MCY)
1    27           ESP        D(ARCTAN)
0    29           STC        PHIV
0    31           NSX                     RETURN FROM PROC .FIG4
```

5.2.5 Figure 5

INPUTS:

$E_{y,p}$ = gain constants

$\varepsilon_{y,p}$ = error signal

OUTPUTS:

δ_{y,p_m} = nozzle command

EQUATION:

24. $$\delta_{y,p_m} = E_{y,p_1}\left\{\left(\varepsilon^{L}_{y,p_m} + E_{y,p_6}\left[\varepsilon_{y,p_m} - \varepsilon_{y,p_{m-1}}\right]\right)\right.$$

$$\left. + E_{y,p_2}\left(\varepsilon^{L}_{y,p_{m-1}} + E_{y,p_6}\left[\varepsilon_{y,p_{m-1}} - \varepsilon_{y,p_{m-2}}\right]\right)\right\}^{L}$$

(Reference 3)

Figure 5

SCALING

Quantity	Maximum Bit Required	Least Bit Required
δ_{y,p_m}	+6	-6
$\epsilon_{y,p}$	+6	-8
$E_{y,p}$	+2	-13

TIMING

Maximum execution time is 150 μsec.

Figure 5 (contd)

5.2.5.1 Figure 5 Analysis

Figure 5 is programmed as an isolated statement in the middle of some other computation. Therefore, there is no procedure return and no waste is counted to fill the last word.

The notation used to state the equation requires some explanation:

a) ε^{L}_{y,p_k} ; k = m-1, m means

$\text{Min} \{\varepsilon_{y,p_k}, \text{Lim1}\}$

b) $\left| \ldots \text{ etc } \ldots \right|^{L}$ means

$\text{Min} \, [\{\ldots \text{ etc } \ldots\} , \text{Lim2}]$

where Liml and Lim2 are globally defined numbers.

Thus, with the following translation of symbols, the equation to be evaluated is that shown in the SPL code for Figure 5.

E_{y,p_i} = EYPi ; i = 1, 2, 6

$\epsilon_{y,p_{m-k}}$ = EPMK ; K = 0, 1, 2

δ_{y,p_m} = DELYPM

Figure 5 illustrates the utility of the MIN operation of the AGC and the .MIN intrinsic for SPL which this report proposes. Figure 5 also shows that the programmer can order statements internally to minimize stack depth.

The statistics for Figure 5 are

	No. Syllables	No. Bytes
FIG 5	26	40

Maximum stack depth is 4 words.

PROGRAM SPL Code: Figure 5 WS-1003-7-2

PROGRAMMER

DATE

```
    .....
DELYPM = .MIN(.MIN(EPM0,LIM1)+(EPM0-EPM1)*EYP6+
             (.MIN(EPM1,LIM1)+(EPM1-EPM2)*EYP6)*EYP2,LIM2)*EYP1
    .....
```

PROGRAM AGC Code: Figure 5 WS-1008-7-2

PROGRAMMER

DATE

Stack Depth	Byte	Label	Operation	Address	Comment
1	0	FIG5	FSS	EPMO	
2	2		FSS	LIM1	
1	4		MIN		.MIN(EPMO, LIM1)
2	5		FSS	EPMO	
3	7		FSS	EPM1	
2	9		−s		
3	10		FSS	EYP6	
2	12		*s		(EPMO-EPM1)*EYP6
1	13		+s		.MIN(0)+(0)*EYP6
2	14		FSS	EPM1	
3	16		FSS	LIM1	
2	18		MIN		.MIN(EPM1, LIM1)
3	19		FSS	EPM1	
4	21		FSS	EPM2	
3	23		−s		
4	24		FSS	EYP6	
3	26		*s		(EPM1-EPM2)*EYP6
2	27		+s		.MIN(1)+(1)*EYP6
3	28		FSS	EYP2	
2	30		*s		( )*EYP2
1	31		+s		ENTIRE LIM2 COMPARAND
2	32		FSS	LIM2	
1	34		MIN		QUANTITY TO * BY EYP1

PROGRAM

PROGRAMMER

DATE

```
2  35       FSS    EYPI
1  31        *S
0  38       SRC    DELYPM
```

5.2.6 Library Function Generators

The following sections deal with the SPL/MK III mathematical functions .SIN, .COS, .ARCTAN and .SQRT. The algorithms used to compute these are based on those used for the IBM 360 Fortran IV library (IBM Publication GC28-6596-4). These have been used because of the compatible word and floating point mantissa length. The algorithms presented here are for single precision. Each algorithm is presented first as an SPL code and then as hand-coded and optimized AGC code based on the SPL logic. The statistics for each algorithm include a comparison with the code length for the same algorithm when a straightforward, non-optimizing compilation approach is used as per the foregoing examples, Figures 1-5. No comparison should be made between the AGC renditions of these algorithms and those for the 360 without making allowance for the difference in operating environment, e.g., error testing and message formatting in the latter.

5.2.6.1 Sine/Cosine Analysis

The sine and cosine are contained in three separate procedures because neither SPL/MK III nor the AGC architecture provides for multiple procedure entries into a program segment. The first two procedures .SIN and .COS merely call the procedure .SINCOS which embodies the common logic. The former two pass on their single arguments together with a second argument which flags whether the sine or cosine is being computed.

The sine and cosine are computed according to the following steps:

1. Let $Z = |X| \div \pi/4$ and let q = integer part of Z,

   r = fractional part of Z.

2. If computing .COS(X), $q \leftarrow q + 2$

   If computing .SIN(X) and $X < 0$, $q \leftarrow q + 4$.

3. Let q_0 = q Mod 8. The following table shows the computation to be performed contingent upon the value of q_0:

q_0	SIGN	FUNCTION	ARGUMENT $\div \pi/4$
0	+	Sin	r
1		Cos	1-r
2		Cos	r
3		Sin	1-r
4	-	Sin	r
5		Cos	1-r
6		Cos	r
7		Sin	1-r

Hence compute $\pm$ sin ($\pi/4$ r_1) or $\pm$ cos ($\pi/4$ r_1) where r_1 = r or 1-r and $0 \leq r_1 \leq 1$.

4. Compute $\sin (\pi/4\ r_1) \cong r_1\ (S_3\ r_1{}^6 + S_2\ r_1{}^4 + S_1\ r_1{}^2 + S_0)$

Compute $\cos (\pi/4\ r_1) \cong \quad C_3\ r_1{}^6 + C_2\ r_1{}^4 + C_1\ r_1{}^2 + C_0$

The steps for computing q and then q_0 embedded in steps 1, 2 and 3 are all performed in a single statement by the use of Boolean relations as arithmetic operators and the logical AND operator, LAND, is used throughout to find remainders Modulo whole powers of 2, e.g. LAND 7 stands for Mod 8, LAND 1 for Mod 2. The quantity .QUO (Q + 1, 2) LAND 1 results in a 0 for each value of Q (i.e., q_0) in the table of step 3 that requires the sine computation and a 1 for each value requiring the cosine. The final computations represent a sacrifice of space to speed, i.e., code compression can be achieved first by looping the computation and second by putting the S and C coefficients into an array so that they may be referred to in a single formula as COEF (K, I) where K = 0, 1, 2, 3 and I is the value .QUO (Q + 1, 2) LAND 1. Finally, the value returned via the formula .SIGN (3 - Q) * Z (9 bytes) saves having to write

```
IF Q LQ 3 THEN RETURN (Z)
                   ELSE RETURN (-Z) END (14 bytes).
```

The statistics for these procedures are as follows:

Constants	- 9 words (8 coefficients and $\pi/4$)
Temporary storage	- 3 words (Z, R and Q)
Maximum stack depth	- 6 words (This can be reduced to 5)

Control Stream

	Optimized			Compiled		
	#Syll	#Bytes	Waste Bytes	#Syll	#Bytes	Waste Bytes
.SIN	10	12	2	6	8	2
.COS				6	8	2
.SINCOS	87	116	3	92	124	0
Totals	97	128	5	104	140	4

Considering total byte count, optimization results in a net saving of 9% in this case. Optimization also saves a total of seven memory accesses for operand fetching or storing.

PROGRAM SPL Code : .SIN/.CØS and .SINCØS .

PROGRAMMER

DATE

```
PRØC  .SIN(X) F 7
EXIT(.SINCØS(X,1))

PRØC  .CØS(X) F 7
EXIT(.SINCØS(X,0))

PRØC  .SINCØS(X,SINE) F 7
       DECLARE F 7,X,Z,R,S3,S2,S1,S0,C3,C2,C1,C0 ''PRESET LAST''
       DECLARE I  ,Q                                ''    EIGHT   ''
       DECLARE BØØLEAN, SINE

       Z = .ABS(X)/PIØV4
       R = .FRAC(Z)
       Q =(.INT(Z)+4*(SINE AND X LS 0)+2*(NØT SINE)) LAND 7

       IF(Q LAND 1)EQ 1 R = 1-R

       Z = R**2

       IF(.QUØ(Q+1,2) LAND 1)EQ 0
       THEN Z = (((S3*Z + S2)*Z + S1)*Z + S0)*R
       ELSE Z =   ((C3*Z + C2)*Z + C1)*Z + C0     END

EXIT(.SIGN(3-Q) * Z)
```

PROGRAM AGC Code : .SIN/.CØS/.SINCØS

PROGRAMMER

DATE

```
1    0  .SIN  LFS   1
2    1        LFS   5
2/1  2        JMP
-    3        NØP
1    4  .CØS  LFS   0         THIS CONCATENATION OF TWO
2    5        FSC   2           SEGMENTS CANNOT BE DONE
2    7        XCH               BY COMPILER --SAVES 1 WORD
1    8        ESP   D(SINCØS)
1   10        NSX
-   11        NØP
```

PROGRAM

PROGRAMMER | DATE

```
1    0. SINCØS FSC   3      }
1    2         ABS          }
2    3         FST   PIØV4  }  Z = .ABS(X)/PIØV4
1    5          ÷RS         }
1    6         SRK   Z      }
1    8         FRAC          } R = .FRAC(Z)  R RETAINED FØR
1    9         STK   R       }                  FUTURE USE
2   11         FSS   Z      }
2   13         INT          }
3   14         LFS   4      }
4   15         FSC   2      }
5   17         FSC   3      }
6   19         LFS   0      }
5   20          LS          }
4   21         AND          }  (.INT(Z)+4(SINE AND X LS 0)
3   22          *S          }
2   23          +S          }           +2(NØT SINE))
3   24         LFS   2      }
4   25         FSC   2      }  LAND 7
4   27         NØT          }
3   28          *S          }
2   29          +S          }
3   30         LFS   7      }
2   31         AND          }
```

PROGRAM

PROGRAMMER DATE

```
2  32          STK   Q    }
3  34          LFS   1    }
2  35          AND        }(Q LAND 1) EQ 1
3  36          LFS   1    }
2  37          EQ         }
3  38          LFS   F1    FETCH ADDRESS ØF FALSE BRANCH
1  40          JØF         JUMP ØN FALSE
2  41          LFS   1     R IS ALREADY ØN TØS
1  42          -S          R-1
1  43          CHS         1-R
1  44          STK   R
2  46  F1      DUP        }
1  47          *S         } Z = R**2
1  48          SRK   Z    } RETAIN Z FØR FURTHER USE
2  50          FSS   Q    }
3  52          LFS   1    }
2  53          +S         }
3  54          LFS   2    }
3  55          ÷I         } (.QUD(Q+1,2) LAND 1) EQ 0
2  56          NIX        }
3  57          LFS   1    }   NIX REMOVES .REM(Q+1,2)
2  58          AND        }
3  59          LFS   0    }
2  60          EQ         }
```

PROGRAM

PROGRAMMER | DATE

```
3    61   LFS   F2      FETCH ADDRESS ΦF ELSE CLAUSE
3/1  63   JΦF           JUMP ΦN FALSE: Z ΦN TΦS
2    64   FSP   S3    ⎫
1    66    *S         ⎪
2    67   FSP   S2    ⎪
1    69    +S         ⎪
2    70   FSS   Z     ⎪      THEN CLAUSE
1    72    *S         ⎪
2    73   FSP   S1    ⎬ (((S3*Z+S2)*Z+S1)*Z+S0)*R
1    75    +S         ⎪
2    76   FSS   Z     ⎪
1    78    *S         ⎪
2    79   FSP   S0    ⎪
1    81    +S         ⎪
2    82   FSS   R     ⎪
1    84    *S         ⎭
2    85   LFS   .BE2    FETCH BEYΦND-ELSE ADDRESS
2/1  87   JMP           JUMP UNCΦNDITIΦNALLY
```

PROGRAM

PROGRAMMER DATE

```
2   88  F2    FSP  C3    }
1   90         *S        }
2   91        FSP  C2    }
1   93         +S        }   ELSE CLAUSE: ENTER WITH Z AT TΦS
2   94        FSS  Z     }
1   96         *S        }   ((C3*Z+C2)*Z+C1)*Z+C0
2   97        FSP  C1    }
1   99         +S        }
2  100        FSS  Z     }
1  102         *S        }
2  103        FSP  C0    }
1  105         +S        }
2  106  BEZ   LFS  3     }
3  107        FSS  Q     }  .SIGN(Z-Q)*TΦS
2  109         -S        }
2  110        SGN        }
1  111         *S        }
1  112        NSX            RETURN FROM .SINCΦS
   113        NΦP        }
   114        NΦP        }WΦRD FILLER
   115        NΦP        }
```

PROGRAM

PROGRAMMER DATE

```
2   88  FL    FSP  C3    }
1   90          *S       }
2   91        FSP  C2    }
1   93          +S       }   ELSE CLAUSE: ENTER WITH Z AT TΦS
2   94        FSS  Z     }
1   96          *S       }  ((C3*Z+C2)*Z+C1)*Z+C0
2   97        FSP  C1    }
1   99          +S       }
2  100        FSS  Z     }
1  102          *S       }
2  103        FSP  C0    }
1  105          +S       }
2  106  BEZ  LFS  3     }
3  107        FSS  Q     }  .SIGN(Z-Q)*TΦS
2  109          -S       }
2  110        SGN        }
1  111          *S       }
1  112        USX            RETURN FRΦM .SINCΦS
   113        NΦP        }
   114        NΦP        }WΦRD FILLER
   115        NΦP        }
```

5.2.6.2 Arctangent Analysis

The steps in the computation of the arctangent are as follows:

1. Reduce argument to the first octant by

   1.1 $.\mathrm{ARCTAN}\ (X) = -.\mathrm{ARCTAN}\ (-X)$ if $X < 0$

   1.2 $.\mathrm{ARCTAN}\ (X) = \pi/2 - .\mathrm{ARCTAN}\ (1/X)$ if $X > 1$

2. Reduce further to $|X| \leq \tan\ (\pi/12)$ by

   $$.\mathrm{ARCTAN}\ (X) = \pi/6 + .\mathrm{ARCTAN}\ ((X\sqrt{3} - 1)/(X + \sqrt{3}))$$

   The value of $\left|\dfrac{X\sqrt{3} - 1}{X + \sqrt{3}}\right| \leq \pi/12$ if $\tan \pi/12 < X \leq 1$

   $X\sqrt{3} - 1$ is computed as $(\sqrt{3} - 1)\ X - 1 + X$ to avoid loss of significant figures.

3. For $|X| \leq \pi/12$

   $$.\mathrm{ARCTAN}\ (X) \cong X\ (A + BX^2 + C/(X^2 + D))$$

   (continued fraction approximation)

   $A = 0.60310579$

   $B = -0.05160454$

   $C = 0.55913709$

   $D = 1.4087812$

To this basic algorithm we have prefixed the following two tests

```
0.1 IF .ABS (X) LS LØWLIM RETURN (X)
0.2 IF .ABS (X) GR UPRLIM RETURN (.SIGN(X) * PIØV2)
```

The values for LOWLIM and UPRLIM used are those suggested by Algorithm 241 of the Collected Algorithms of the ACM.

This algorithm lends itself well to the AGC's ability to handle recursive entries to a procedure provided that the only arguments are input arguments represented by value. The resultant SPL and AGC code is very compact. In execution, the maximum build up in the stack is four levels:

- the originating call from outside;
- up to three internal calls of .ARCTAN on itself.

Each level adds three words to the stack, the argument plus two RPW's.

The statistics for the .ARCTAN procedure are as follows:

Constants	-	11 words (9 local plus $\pi/2$ and $\pi/6$)
Temporary storage	-	0 words
Maximum stack depth	-	4 words plus variable stacking due to recursion.

Control Stream

	Optimized			Compiled		
	#Syll	#Bytes	Waste Bytes	#Syll	#Bytes	Waste Bytes
.ARCTAN	76	96	2	79	112	0

Considering total byte count, optimization results in a net saving of 14% in this case. Optimization also saves a total of 12 (13 vs 25) memory accesses for operand fetching and storing. This is done by judicious use of the DUP and XCH operators.

PROGRAM SPL CODE: .ARCTAN

PROGRAMMER

DATE

```
PRØC .ARCTAN(X) F 7 DECLARE F 7, X
       DECLARE F 7 CØNSTANT, LØWLIM =2.13E-22    ,A= 0.60310579,
                             UPRLIM =1.34E8      ,B=-0.05160454,
                ''TAN PI/12=''TN15   =0.26794919 ,C= 0.55913709,
                             RT3M1   =0.73205081 ,D= 1.4087812 ,
                             RT3     =1.7320508

       IF .ABS(X) LS LØWLIM RETURN(X)
       IF .ABS(X) GR UPRLIM RETURN(.SIGN(X)*PIØV2)

       IF X LS 0       RETURN(-.ARCTAN(-X))

       IF X GR 1       RETURN(-.ARCTAN(1/X)+PIØV2)

       IF X GR TN15 RETURN(.ARCTAN((RT3M1*X-1+X)/(X+RT3))+PIØV6)

EXIT(X*(A+C*X**2+C/(X**2+D)))
```

PROGRAM AGC Code: .ARCTAN

PROGRAMMER

DATE

Stack Depth	BYTE	LABEL	OPERATION	ADDRESS	COMMENT
1	000.	.ARCTAN	FSC	2	TΦS = X
2	2		DUP		X, X
2	3		ABS		|X|, X
3	4		FSP	LΦWLIM	L, |X|, X
2	6		LS		L > |X|, X
3	7		LFS	T1	ADDRESS TRUE BRANCH 1, L > |X|, X
3/1	8		JΦT		JUMP ΦN TRUE
2	9		DUP		TΦS = X, X
2	10		ABS		|X|, X
3	11		FSP	UPRLIM	U, |X|, X
2	12		GR		U < |X|, X
3	14		LFS	F2	ADDRESS FALSE BRANCH 2, U < |X|, X
3/1	15		JΦF		JUMP ΦN FALSE
1	16		SGN		TΦS = ±1
2	17		FST	PIΦV2	π/2, ±1
1	19		*S		±π/2
1	20	T1	NSX		TΦS = ±π/2 ΦR X RETURN
2	21	F2	DUP		X, X
3	22		LFS	0	0, X, X
2	23		LS		0 > X, X
3	24		LFS	F3	ADDRESS FALSE BRANCH 3, 0 > X, X
3/1	25		JΦF		JUMP ΦN FALSE
1	26		CHS		TΦS = -X
1	27		ESP	D(ARCTAN)	.ARCTAN(-X)

PROGRAM

PROGRAMMER DATE

```
1  029        CHS        TΦS=-.ARCTAN(-X)
1   30        NSX        RETURN
2   31  F3    DUP        TΦS=X,X
3   32        LFS   1        1,X,X
2   33         GR            1<X,X
3   34        LFS   F4   ADDRESS FALSE BRANCH 4,1<X,X
2/1 36        JΦF        JUMP ΦN FALSE
2   37        LFS   1    TΦS=1,X
2   38        XCH            X,1
1   39         ÷KS           1/X
1   40        ESP   D(ARCTAN)  .ARCTAN(1/X)
1   42        CHS            -.ARCTAN(1/X)
2   43        FST   PIΦV2    π/2,-.ARCTAN(1/X)
1   45         +S            π/2 -.ARCTAN(1/X)
1   46        NSX        RETURN
2   47  F4    DUP        TΦS=X,X
3   48        FSP   TNI5     TNI5,X,X
2   50         GR            TNI5<X,X
3   51        LFS   F5   ADDRESS FALSE BRANCH 5,TNI5<X,X
2/1 53        JΦF        JUMP ΦN FALSE
2   54        DUP        TΦS=X,X
3   55        DUP            X,X,X
4   56        FSP   RT3MI    RT3MI,X,X,X
3   58         *S            RT3MI*X,X,X
```

PROGRAM AGC CODE: .ARCTAN

PROGRAMMER

DATE

2	059		+S		TΦS=RT3M1*X+X, X
3	60		LFS	1	1, RT3M1*X+X, X
2	61		−S		RT3M1*X+X−1, X
2	62		XCH		X, RT3M1*X+X−1
3	63		FSP	RT3	RT3, X, RT3M1*X+X−1
2	65		+S		RT3+X, RT3M1*X+X−1
1	66		÷RS		(RT3M1*X+X−1)/(RT3+X)
1	67		ESP	D(ARCTAN)	.ARCTAN(ETC)
2	69		FST	PIΦV6	π/6, .ARCTAN(ETC)
1	71		+S		π/6+.ARCTAN(ETC)
1	72		NSX		RETURN
2	73	F5	DUP		TΦS=X, X
3	74		DUP		X, X, X
2	75		*S		X^2, X
3	76		DUP		X^2, X^2, X
4	77		FSP	B	B, X^2, X^2, X
3	79		*S		B*X^2, X^2, X
4	80		FSP	A	A, B*X^2, X^2, X
3	82		+S		A+B*X^2, X^2, X
3	83		XCH		X^2, A+B*X^2, X
4	84		FSP	D	D, X^2, A+B*X^2, X
3	86		+S		D+X^2, A+B*X^2, X
4	87		FSP	C	C, D+X^2, A+B*X^2, X
4	89		XCH		D+X^2, C, A+B*X^2, X

PROGRAM

PROGRAMMER DATE

```
3  070      ÷RS    TØS = C/(D+X^2), A+B*X^2, X
2   71      +S           C/(D+X^2)+A+B*X^2, X
1   72      *S           (C/(D+X^2)+A+B*X^2)*X
1   73      NOX    RETURN
    74      NOP    } WORD FILLER
    75      NOP    }
```

5.2.6.3 Square Root Analysis

The square root is computed according to the steps of the following algorithm:

1. If $X = 0$, return an answer of 0.

2. Let $X = 2^{2p} \cdot m$ where p is an integer and $1/4 \leq m < 1$. Since X may be an integer which is generally unnormalized, special attention is required for this case.

3. Then $\sqrt{X} = 2^p \cdot \sqrt{m}$

4. As a first approximation to $\sqrt{m}$, one of the following hyperbolic forms is used:

$$\sqrt{m} \doteq a_i + b_i/(c_i + X)$$

$$i = 0 \text{ for } 1/4 \leq m < 1/2$$

$$i = 1 \text{ for } 1/2 \leq m < 1$$

The values of these constants are not specified in the coding which follows.

5. Define $y_0 = 2^p \cdot (a_i + (b_i)/(c_i + X))$

Perform two Newton-Raphson iterations on y_0:

$$y_{n+1} = 1/2 \ (y_n + (y_n + X/(y_n)))$$

and return y_2 as the answer.

The coding assumes that the normalization called for in step 2 can be accomplished by a floating point addition of a floating point zero (non-zero exponent, zero mantissa). Addition of a true zero will leave an integer unnormalized.

Since the AGC regards the radix point as being at the right hand of the mantissa, a normalized AGC floating point number is actually

$$2^e \cdot m \times 2^{24}$$

Therefore, to recast X as $2^{2p} \cdot m \times 2^{24}$ one must set

$p = [(e-24)/2]$ when e is even and

$p = [(e-23)/2]$ when e is odd.

In the latter case, one must also shift m right one place. This is done by .QUO(MANT, 2). The hyperbolic approximations are rendered in integer arithmetic before the exponent is reinserted. Finally, the two applications of the Newton-Raphson iteration are combined into one formula.

$$.5 \left[.5 \left(Y + \frac{X}{Y} \right) + \frac{X}{.5\,(Y + X/Y)} \right]$$

$$= (Y + X/Y)/4 + X/(Y+X/Y)$$

The DUP and XCH operators are used effectively to avoid computing the common factor Y + X/Y twice. In practice, when hand-coded optimization is not available to the programmer, he can achieve some of the same effect by making a chained assignment out of such a formula:

TEMP/4 + X/TEMP = Y + X/Y

The statistics for the .SQRT procedure are:

Constants -	9 words (6 coefficients, 2 HEX literals, 1 descriptor)
Temporary Storage-	1 word (MANT)
Maximum stack depth -	5 words

Control Stream

Optimized			Compiled		
#Syll	# Bytes	Waste Bytes	# Syll	# Bytes	Waste Bytes
63	76	3	68	100	3

The significant result of optimization in this case besides a 24% reduction in length is the reduction of the number of memory accesses for operand fetches and stores:

Compiled - 28

Optimized - 10

Again this has been done by using DUP and XCH to hold values in the stack until they are needed thus avoiding many stores and fetches.

PROGRAM SPL Code: .SQRT

PROGRAMMER DATE

```
PROC  .SQRT(X) F 7
       DECLARE F 7, X, Y, XN
       DECLARE   I, XPØN, MANT, K,
                     CØEF(3, 2) CØNSTANT=(A1, B1, C1, A0, B0, C0)
       ØVERLAY       Y=MANT, XN=K

       IF X LQ 0  RETURN(X)

       MANT = HEX'00FFFFFF' LAND XN = X + HEX'31000000'
       XPØN = XN(S 1//7)-24
       K    = XPØN LAND 1

       XPØN = .QUØ(K+XPØN, 2)
       MANT = CØEF(0, K)+.QUØ(CØEF(1, K), CØEF(2, K)+.QUØ(MANT, K+1))

       Y(S 1//7) = XPØN

       Y = Y + X/Y

EXIT(Y/4+X/Y)
```

PROGRAM AGC Code: .SQRT

PROGRAMMER

DATE

```
                                                                   BYTE  STACK
                                                                         DEPTH
.SQRT  FSC    2       TØS=X                                        000   1
       DUP                X,X                                        2   2
       LFS    0           0,X,X                                      3   3
        LQ                0>X,X                                      4   2
       LFS    8           8,0>X,X                                    5   3
       JØF                X                                          6   1
       NSX            RETURN WITH .SQRT(X)=X, X≤0                    7   1
       FSP    H31     TØS=X'31000000',X                              8   2
        +S               =X NØRMALIZED ØR XN                        10   1
       DUP                XN,XN                                     11   2
       FSP    HFF         X'00FFFFFF',XN,XN                         12   3
       AND                MANT,XN                                   14   2
       STC    MANT        XN                                        15   1
       LFS    1           1,XN                                      17   2
       LFS    7           7,1,XN                                    18   3
       SXT                XN(S 1//7)                                19   1
       LFS    24          24,XN(S 1//7)                             20   2
        -S                XPØN                                      21   1
       DUP                XPØN,XPØN                                 22   2
       LFS    1           1,XPØN,XPØN                               23   3
       AND                K,XPØN                                    24   2
       DUP                K,K,XPØN                                  25   3
       SCV                K,XPØN AND K --> CVR                      26   2
        +S                K+XPØN                                    27   1
```

PROGRAM	AGC Code: .SQRT
PROGRAMMER	DATE

			BYTE	STACK DEPTH
LFS	2	TØS=2, K+XPØN	028	2
÷i		REM, QUØ=XPØN	29	2
NIX		XPØN	30	1
LFS	1	1, XPØN	31	2
FCV		K, 1, XPØN	32	3
ELF	D(CØEF)	CØEF(1, K), YPØN	33	2
FSS	MANT	MANT, C(1, K), XPØN	35	3
FCV		K, MANT, C(1, K), XPØN	37	4
LFS	1	1, K, MANT, C(1, K), XPØN	38	5
÷s		1+K, MANT, C(1, K), XPØN	39	4
÷i		REM, QUØ, C(1, K), XPØN	40	4
NIX		QUO, C(1, K), XPØN	41	3
LFS	2	2, QUØ, C(1, K), XPØN	42	4
FCV		K, 2, QUØ, C(1, K), YPØN	43	5
ELF	D(CØEF)	C(2, K), QUØ, C(1, K), XPØN	44	4
+s		C(2, K)+QUØ, C(1, K), XPØN	46	3
÷i			47	3
NIX		MANT-C(0, K), XPØN	48	2
LFS	0		49	3
FCV			50	4
ELF	D(CØEF)	C(0, K), MANT-C(0, K), XPØN	51	3
+s		MANT, XPØN	53	2
LFS	1	1, MANT, XPØN	54	3
LFS	7	7, 1, MANT, XPØN	55	4

PROGRAM

PROGRAMMER

DATE

				BYTE	STACK DEPTH
	SNS		TØS = Y	056	1
	DUP		Y , Y	57	2
	FSC	2	X , Y , Y	58	3
	XCH		Y , X , Y	60	3
	÷RS		X/Y , Y	61	2
	+S		Y + X/Y CALL THIS Y	62	1
	DUP		Y , Y	63	2
	LFS	4	4 , Y , Y	64	3
	÷RS		Y/4 , Y	65	2
	XCH		Y , Y/4	66	2
	FSC	2	X , Y , Y/4	67	3
	XCH		Y , X , Y/4	69	3
	÷RS		X/Y , Y/4	70	2
	+S		Y/4 + X/Y = .SQRT (X)	71	1
	NSX		RETURN FRØM PRØC .SQRT	72	1
	NØP			73	
	NØP		WØRD FILLER	74	
	NØP			75	

5.2.7 IDCU Analysis

Figures 1 and 5 of CIRAD Report WS-1008-7-2 have been programmed for the IDCU. The coding conventions employed in these two programs are explained in the introductory commentary of Figure 1.

Figure 1 has been programmed with the computation of th: P_n^m's in-line rather than as a procedure. Figure 1 also requires the computation of a rounded quotient with the scaling (9.22), i.e., 22 fractional bits. This is done by the procedure DIV22. The sine/cosine routine accepts arguments in degrees scaled (18.13) and returns results scaled (5.26). These different accuracy requirements necessitate using a different algorithm than was used for the AGC (degree 4 vs degree 3). The algorithm is based on the Chebychev polynomial approximation 4.3.104 of the Handbook of Mathematical Functions (National Bureau of Standards, Applied Mathematics Series 55).

The coding for Figure 5 is entirely straightforward with the common subexpressions

$$\left\{ \epsilon^{L}_{y,p_k} + E_{y,p_6} \left(\epsilon_{y,p_k} - \epsilon_{y,p_{k-1}} \right) \right\}^{L} \quad ; k = m,\ m - 1$$

being evaluated by a close.

The line-by-line commentary for each figure is relatively complete and aids in keeping track of the stack contents and scaling during the computations.

The statistics for these two figures are as follows:

			No. Words					
		No. Instr	Instr.	Const. Var. Temp Store	Literals	Total	No SKP's	Max Stack Depth
Figure 1	G	150	75	31	8	114	21	24
	DIV22	23	10	1	1	12	2	4
	SIND/COSD	54	26	6	6	38	8	7
Figure 5		51	23	9	1	33	3	12

PROGRAM IDCU CODE: FIGURE 1 WS-1008-7-2

PROGRAMMER

DATE

```
                    THE IDCU CØDE PRESENTED ØN THE FØLLØWING
                    PAGES USES CØNVENTIØNS ROUGHLY BASED ON
                    ASM FØR THE UNIVAC 1108. THUS, FØR EXAMPLE,

                    (GCO,GCI) CALLS FØR THE ADDRESS OF A
                              LITERAL IN WHICH ADDRESSES GCO &
                              GCI ARE STØRED IN THE LEFT AND
                              RIGHT HALVES RESPECTIVELY.

                    DØ N X    MEANS REPEAT THE CØDING X N TIMES

                    LØD,LH    MEANS LØAD RIGHT
                    LØD,RH                LEFT HALF ØF WØRD

                    LDX,N     MEANS LØAD INDEX REGISTER N

                    ØTHER USAGES ARE MØRE ØR LESS TRANSPARENT.

                    THE PSEUDØ-ØP BYT CALLS FØR ASSEMBLY TØ
                    BEGIN AT THE BYTE NAMED IN THE ADDRESS FLD.

                    PROGRAM STARTS IN BYTE 2 TØ MINIMIZE
                            NUMBER ØF SKP'S
```

PROGRAM

PROGRAMMER | DATE

```
                                                                   WORD
.     BYT     0           . DATA AREA ØN WØRD BØUNDARY
A0    DØ      6 +0        . SEVEN LØCATIØNS                       *000
A6    +       0           .       TØ HØLD CØMPUTED P(M,N)'S.      *  6
AER   +       0           .                                       *  7
NPØW  +       0           .                                       *  8
      +       0           . SUM                                      9
      +       0           . GUM                                     10
CGFAC +       0           . CG1/(R*R*CØS(PHI)) (4.27)             * 11
      +       0           . H                                       12
      +       1,0         . N                                       13
LADR  +       LMN-1,0     . LMN ARRAY ADDRESS-1 IN LEFT             14
JADR  +       JMN-1,0     . JMN ----- --------- -- ----             15
LAMDA +       0           . LAMDA PRESTORED HERE (18.13)            16
PHI   +       0           . PHI     .         .  (18.13)            17
CG1   +       0           . CG1     .         .  ( 9.22)            18
R     +       0           . R       .         .  ( 9.22)            19
AE    +       0           . AE      V         V  ( 9.22)            20
ONES  DØ      6 +1*/26    . SEVEN ØNES                              21
      +       1*/26       .         SCALED  (5.26)                  27
CPH   +       0           . CØS(PHI) (5.26)                       * 28
SPH   +       0           . SIN(PHI) (5.26)                       * 29
.                         * NØTE MØST ØF THIS DATA IS ALL
.                           MØVED TØ STACK. ØNLY ITEMS STARRED
.                           IN CØLUMN 60 ARE ALTERED IN MEMØRY
```

PROGRAM

PROGRAMMER

DATE

```
. ACCUMULATED LIST OF LITERALS
. IN ORDER OF OCCURRANCE
.
. (G0,G1)
. (G2,G3)
. (4,0)
. (1,0)
. (G7,0)
. (G8,G9)
. (G12,G13)
. (5,0)
```

PROGRAM

PROGRAMMER DATE

```
                                                                BYTE  PTR
G*    BYT     2           . ENTER W JMP, TΦS=RETURN ADDR                 0
      SLL     AE,4        . TΦS=PHI,CGI,R,AE,RA                   002    4
      BLT,U   0           . CΦPY PHI (18.13) TΦ TΦS                 6    5
      LΦD,LH  (G0,G1)     . SET RETURN ADDRESS                      8    6
      JMP     SIND        . SINE, ARGUMENT IN DEGREES              11    6
      SKP                 . SCALING ΦF RESULT IS (5.26)            14  ...
G0    PDP                 . TΦS=SPH,PHI,ETC                        15    5
      STΦ     SPH         . TΦS=PHI,CGI,ETC                        16    4
      LΦD,RH  (G0,G1)     . SET RETURN ADDRESS                     19    5
      SKP                 . CAN'T JUMP FRΦM BYTE 2                 22    5
      JMP     CΦSD        . CΦSINE ARGUMENT IN DEGREES             23    5
G1    PDP                 . TΦS=CPH,CGI,ETC                        26    4
      STN     CPH         . CPH ASSUMED NΦN-ZERΦ                   27    4
      SHA,L   2           . CPH(5.26)-->CPH(3.28)                  30    4
      BLT,U   2           . TΦS=R(9.22)                            32    5
      SHA,L   5           . R(9.22)-->R(4.27)                      34    5
      BLT,U   0           . TΦS=R(4.27),R(4.27),CPH(3.28           36    6
      DΦ      2 BLT,U 5   . TΦS=R(9.22),AE(9.22),R(4.27)           38    8
      SKP                 . CAN'T LΦD FRΦM BYTE 2                  42    8
      LΦD,LH  (G2,G3)     . SET RETURN ADDRESS                     43    9
      SKP                 . CAN'T JMP FRΦM BYTE 2                  46    9
      JMP     DIV22       . GET AE/R (9.22) RΦUNDED=AER            47    9
G2    PDP                 . TΦS=AER(9.22)                          50    7
      SHA,L   5           . AER(9.22)-->AER(4.27)                  51    7
```

PROGRAM

PROGRAMMER   DATE

```
                                                                    BYTE  PTR
      STN        NPØW        . STØRE AS NPØW, AER                    053   7
      STØ        AER         . TØS=R(4.27), R(4.27), CPH(3.28         56   6
      MPR                    . TØS=R*R(9.22), CPH(3.28)               59   5
      SHA, L     4           . R*R(9.22)-->R*R(5.26)                  60   5
      MPR                    . TØS=R*R*CPH(9.22), CGI(9.22)           62   4
      LØD, RH    (G2, G3)    . SET RETURN ADDRESS                     63   5
      SKP                    . GET CGI/(R*R*CPH) (9.22)               66   5
      JMP        DIV22       .   AS ROUNDED QUØTIENT                  67   5
G3    PDP                    .   CALL THIS CGFAC                      70   3
      SHA, L     5           . CGFAC(9.22)-->CGFAC(4.27)              71   3
      STØ        CGFAC       . END ØF PRELIM CØMPUTATIØNS             73   2
      DØ         2 PDP       . LØAD STACK 4 MAJØR ITERATIØN           76   0
      SLL        LAMDA, 8    . TØS=SUM, GUM, CGFAC, M-1, N-1,         78   8
      SKP                    .       LADR-1, JADR-1, LAMDA            82   8
      LDX        (4, 0), XR2 . MAJØR ITERATIØN 5 TIMES                83   8
G4    BLT, U     4           . TØS=N-1(15.16)                         86   9
      STN, LH    0403        . STACK-->XR3 CENTRAL LØCATIØN           88   9
      ADM        (1, 0)      . TØS=N(15.16)                           91   9
      BLT, D     5           . STØRE N IN STACK                       94   9
      PDP                    .                                        96   8
      LØD        (1, 0)      . INITILIZE M TØ 1                       97   9
      JMP        G6          .                                       100   9
G5    BLT, U     3           . TØS= M(15.16)                         103   9
      ADM        (1, 0)      . M+1--> M(15.16)                       105   9
```

PROGRAM

PROGRAMMER | DATE

				BYTE	PTR
G6	BLT, D	4	. STØRE 11 IN STACK	108	9
	SHA, L	11	. TØS = M(4.27)	110	9
	BLT, U	6	. ADDRESS-1 ØF L(M,N)	112	10
	SKP		.	114	10
	ADM	(1,0)	. ADDR ØF CURRENT L(M,N)	115	10
	BLT, D	7	. SAVE IT	118	10
	ILL		. TØS = L(M,N) (18.13)	120	10
	BLT, U	9	. TØS = LAMPA (18.13)	121	11
	SUB		. TØS = LAMDA-L(M,N) (18.13)	123	10
	SHA, L	9	. SCALE SAME AT (9.22)	124	10
	BLT, U	1	. TØS = M(4.27), LAM-LMN(9.22)	126	11
	SHA, R	4	. M(8.23)	128	11
	MPR		. M(LAM-LMN) (18.13)	130	10
	LØD	(G7,0)	. SET RETURN ADDRESS	131	11
	SKP		.	134	11
	JMP	SIND	. SIN(DEGREES) (5.26)	135	11
G7	PDP		.	138	10
	LØD	(G8,G9)	. TØ G8 FIRST TIME	139	11
	IJP		. TØ G9 THEREAFTER	142	11
G8	SHL, L	16	. RØTATE AND	143	11
	STØ	(G8,G9)	. STØRE SWITCH	145	10
	SLL	AE, 7	. LØAD STACK WITH P(M,N-1)	148	17
	SLL	ØNES, 7	. P(0,N) = ALL 1 WHEN N=2	152	24
	JMP	G11	.	156	24

PROGRAM

PROGRAMMER DATE

```
                                                                           BYTE     PTR
G9    PDP                        . ENTER HERE M>2                             159   10
      LΦP     (G10,G13)          . TΦ G10 WHENEVER M=1                        160   11
      IJP                        . ELSE TΦ G13                                163   11
G10   SHL,L   16                 . RΦTATE AND                                 164   11
      SKP                        .            STΦRE SWITCH                    166   11
      STΦ     (G10,G13)          .                                            167   10
      SLL     AE,7               . LΦAD STACK WITH P(M,N-1)                   170   17
      SLL     A6,7               . AND P(0,N)=1 WHEN N>2                      174   24
      SKP                        .                                            175   24
G11   LDX     (5,0),XR1          . SET CΦUNT=6 FΦR P(M,N) LΦΦP              K 179   24
G12   BLT,D   7                  . SAVE VALUE ΦF P(1.30)               182  24...   19
      LΦD     SPH                . SPH(5.26)                           184  25...   20
      MPR                        . P(M-1,N)*SPH (7.24)                 187  24...   19
      SHA,L   6                  . P(M-1,N)*SPH (1.30)                 188  24...   19
      XCH                        .                                     190  24...   19
      LΦD     CPH                . CPH(5.26)                           191  25...   20
      MPR                        . P(M,N-1)*CPH (7.24)                 194  24...   19
      SHA,L   6                  .   DITTΦ      (1.30)                 195  24...   19
      ADD                        . P(M,N) (1,30)                       197  23...   18
      INV     G12,XR1,1          . 5-->4-->3-->2-->1-->0               198  23...   18
      BLT,D   7                  . STΦRE FINAL VALUE P(6,N)                   202   18
      PDP                        .                                            204   17
      SKP                        .                                            205   17
      SLS     A0,7               . STΦRE LIST P(0,N)...P(6,N)                 206   10
```

PROGRAM

PROGRAMMER | DATE

```
                                                                        BYTE  PTR
      PUP                . ADJUST PΦINTER                               210   11
G13   PDP                .                                              211   10
      BLT,U    5         . TΦS=M(15.16)                                 212   11
      SKP                .                                              214   11
      STΦ,LH   0404      . M--> XR1 CENTRAL LΦCATION                    215   10
      SKP                .                                              218   10
      LΦD      A0,XR1    . TΦS=P(M,N)(1.30)                             219   11
      SHA,R    3         .     P(M,N)(4.27)                             222   11
      BLT,U    9         . ADDRESS-1 ΦF J(M,N)                          224   12
      SKP                .                                              226   12
      ADM      (1,0)     . ADDR ΦF CURRENT J(M,N)                       227   12
      BLT,D    10        . SAVE IT                                      230   12
      ILL                . TΦS=J(M,N)(9.22)                             232   12
      SHA,L    5         .     J(M,N)(4.27)                             233   12
      MPR                . J(4.27)*P(4.27)=J*P(9.22)                    235   11
      SHA,L    6         . J*P(3.28)                                    236   11
      MPR                . J*P(3.28)*SIN(5.26)                          238   10
      SHA,L    5         . J*P*SIN(7.22)-->SAME(4.27)                   239   10
      MPR                . J*P*SIN(4.27)*M(4.27)                        241    9
      SHA,L    5         . TERM(9.22)-->TERM(4.27)                      242    9
      ADD                . SUM=SUM+TERM(4.27                            244    8
      SKP                .                                              245    8
      INV      G5,XR3,1  . N-1-->N-2-->...1-->0                         246    8
      SLL.     NPOW,2    . TΦS=AER,NPOW                                 250   10
```

PROGRAM

PROGRAMMER   DATE

```
                                                                          BYTE  PTR
      MPR                    . NPΦW=NPΦW*AER(9.22)                        254   9
      SHA,L      5           . NPΦW(4.27)                                 255   9
      STN        NPΦW        .                                            257   9
      MPR                    . SUM*NPΦW(9.22)                             260   8
      SHA,L      5           . SUM*NPΦW(4.27)                             261   8
      ADD                    . GUM=GUM+SUM*NPΦW (4,27                     263   7
      LΦD        (G10,G13)   . RESET AND                                  264   8
      SHL,L      16          .           STΦRE                            267   8
      STN        (G10,G13)   .                SECΦND SWITCH               269   8
      CLR                    . RESET SUM TΦ 0                             272   8
      SKP                    . CANT INV FRΦM BYTE 1                       273   8
      INV        G4,XR2,1    . 4 --> 3 --> 2 --> 1 --> 0.                 274   8
      SKP                    .                                            278   8
      LΦD        (G8,G9)     . RESET AND                                  279   9
      SHL,L      16          .           STΦRE                            282   9
      STΦ        (G8,G9)     .                FIRST SWITCH              - 284   8
      PDP                    .                                            287   7
      MPR                    . G(9.22)=GUM(4.27)*CGFAC(4,27               288   6
      BLT,D      5           . STΦRE G AT REL 1 IN STACK                  289   6
      DΦ         5 PDP       . MΦVE PΦINTER DΦWN 5 PLACES                 291   1
      XCH                    . TΦS= RETURN ADDRESS IN LEFT                296   1
      IJP                    . RETURN                                     297
      END                    . END ΦF G
```

PROGRAM

PROGRAMMER DATE

```
DIV22*  BYT     0         . DIVIDEND (9.22) AT STACK 0
                          . DIVISØR  (9.22) AT STACK 1
                          . RETURN ADDR (15.16) STACK 2
                          . ASSUMES NEITHER IS = 0
                          . RETURNS RØUNDED QUØTIENT SCALED
                          . (9.22) AT STACK 0, PTR AT STACK 1
                          .
                          . REQUIRES 1 TEMPØRARY LØCATIØN SHFT
                          .          FØR STØRING SHIFT CØUNT
                          .      AND 1 LITERAL (7,0)
```

PROGRAM

PROGRAMMER DATE

				BYTE	PTR
	XCH		. TØS = DIVISØR, RA, DIVIDEND	000	2
	NRM		. TØS = Z(DIVISØR), N(DIVISØR)	1	3
	SKP		. Z( ) = ZERØ CØUNT, N( ) = NØRM #	2	3
	STØ	SHFT	. Z( ) IS SCALED (15.16)	3	2
	BLT, U	2	. TØS = DIVIDEND	6	3
	NRM		. =Z(DIVIDEND), N(DIVIDEND)	8	4
	SBM	SHFT	. =Z(DVD)-Z(DVS)	9	4
	CØM, 2		. =Z(DVS)-Z(DVD)	12	4
	ADM	(7,0)	. = 7+Z(DVS)-Z(DVD)	13	4
	STØ	SHFT	. TØS=N(DVD), N(DVS)	16	3
	SHA, R	1	. MAKE DIVIDEND < DIVISØR	19	3
	DIV		. TØS = QUØTIENT	21	2
	SKP		.	22	2
	LØD	SHFT	. TØS=SCALEFACTØR, QUØTIENT	23	3
	ASR, R		. TØS = QUØTIENT (8.23)	26	2
	INC		. RØUND QUØTIENT	27	2
	BRP	DDO	. .	28	2
	DEC		. .	31	2
	DEC		. V	32	2
	SHA, R	1	. TØS=QUØTIENT(9.22)	33	2
	BLT, D	2	. STØRE RESULT IN STACK 0	35	2
	PDP		. TØS = RETURN ADDRESS	37	1
	IJP		. RETURN	38	1
	END		. END ØF DIV22		

PROGRAM

PROGRAMMER | DATE

```
CØSD* BYT      O         . SINE AND COSINE
SIND* EQU      SNTRY     . ARGUMENT IN DEGREES SCALED (18.13)
                         . IS STØRED IN STACK O
                         . RETURN ADDRESS (15.16) IS STØRED IN
                         . STACK 1
                         .
                         . RETURN WITH RESULT (5.26) IN STACK O
                         . AND PØINTER AT STACK 1.
```

PROGRAM

PROGRAMMER DATE

```
                                                                     WORD
      BYT         0           . DATA KEEP ON WORD BOUNDARY

T     +           0           . HOLDS 2*ARG**2-1                      000
      +           0.000009480*/30 . COEFFICIENTS                        1
      -           0.000547288*/30 .    .                                2
      +           0.018226552*/30 .    .                                3
      -           0.284851033*/30 .    .                                4
B0    +           1.267162131*/30 .    Y                                5
```

PROGRAM

PROGRAMMER

DATE

```
                         . ACCUMULATED LIST ØF LITERALS
                         . IN ØRDER ØF ØCCURRANCE
                         .
                         . (90*/13) I.E., 90(18.13)
                         . (90*/24)
                         . (4*/28)
                         . (SC1,0)
                         . (2*/28)
                         . (3,0)

                         . (1*/30) IS SAME AS (4*/28)
```

PROGRAM

PROGRAMMER

DATE

```
                                                                   BYTE  PTR
       XCH                    . TØS=X,RA  CØSINE ENTRY              000   1
       SBM    (90*/13)        . X-90 --> X                            1   1
       XCH                    . TØS=RA,X                              4   1
SNTRY  XCH                    . TØS=X,RA  SINE ENTRY                  5   1
       SHA,L  1               . X(17.14)                              6   1
       LØD    (90*/24)        . TØS=90(7.24),X(17.14)                 8   2
       XCH                    . TØS=X(17.14),90(7.24)                11   2
       DIV                    .     =X/90(10.21)                     12   1
       SHA,R  1               . SHAKE ØFF SPURIØUS 1 (11.20)         13   1
       SHA,L  9               . REDUCE INTEGER MØD 4 (2.29)          15   1
       SHA,R  1               . ALLØW RØØM TØ ADD 4 (3.28)           17   1
       BRP    SCO             . BRANCH IF PØSITIVE                   19   1
       SKP                    .                                      22   1
       ADM    (4*/28)         . IF NEGATIVE ADD 4 TØ MAKE +          23   1
SCO    BLT,U  0               . TØS=ARG,ARG BØTH (3.28)              26   2
       SHA,R  12              . TØS=ARG(15.16),ARG(3.28)             28   2
       SKP                    .                                      30   2
       ADM    (SC1,0)         . CØNSTRUCT QUADRANT SWITCH            31   2
       IJP                    . JUMP ØN QUADRANT SWITCH              34   2
SC1    JMP    SC3             . QUADRANT 1: LEAVE AS IS              35   2
       SKP                    . NEVER EXECUTED                       38   ...
       JMP    SC2             . QUADRANT 2: SEE QUADRANT 3           39   2
       SKP                    .                                      42   ...
       JMP    SC2             . QUADRANT 3: SUBTRACT FRØM 2          43   2
```

PROGRAM

PROGRAMMER | DATE

```
                                                                                         BYTE PTR
      SKP                 .                                                              046 ...
      PDP                 . QUADRANT 4: SUBTRACT 4                                        47  1
      SBM      (4*/28)    . TΦS=4-ARG(3.28)                                               48  1
      CΦM,2               .     =ARG-4(3.28)                                              51  1
      JMP      SC4        .                                                               52  1
SC2   PDP                 .                                                               55  1
      SBM      (2*/28)    . TΦS=2-ARG(3.28)                                               56  1
      PUP                 . SAVES A JMP                                                   59  2
SC3   PDP                 , TΦS=ARG(3.28) ADJUSTED                                        60  1
SC4   BLT,U    0          .     =ARG, ARG BOTH (3.28)                                     61  2
      SHA,L    3          .                                                               63  2
      BLT,U    0          .     =ARG(0,31) TWICE, ARG(3.28)                               65  3
      MPR                 .     =X**2(1.30), ARG(3.28)                                    67  2
      SHA,L    1          .     =2*XSQ                                                    68  2
      SKP                 .                                                               70  2
      SBM      (1*/30)    .                                                               71  2
      CΦM,2               .     =2*XSQ-1                                                  74  2
      STΦ      T          . SAVE T=2*XSQ-1 (1.30)                                         75  1
      SLL      B0,5       . LΦAD CΦEFFICIENTS                                             78  6
      SKP                 .                                                               82  6
      LDX      (3,0),XR3  . SET ITERATIΦN CΦUNT                                           83  6
      SKP                 .                                                               86  6
```

PROGRAMMER DATE

```
                                                               BYTE
 1 2 3 4 5 6 7 ... 15 16 ... 20 21 ... 45                     46 ... 50 51 ... 55 56 ... 63
.                                                                       XR3  =    3 2 1 0
SC5    LØD       T           . TØS= T, PARTIAL, B'S (1.30       087        /     7 6 5 4
       MPR                   .     = T*PARTIAL (3.28)            90      /       6 5 4 3
       SHA,L     2           .                   (1.30)          91   PTR        6 5 4 3
       ADD                   .     = T*PARTIAL +BNXT, B'S        93       \      5 4 3 2
       INV       SC5, XR3, 1 .                                   94       .\     5 4 3 2
       MPR                   . TØS= SUM(1.30)*X(3.28)            98                    1
       XCH                   . TØS= RA, RESULT                   99
       IJP                   . RETURN                           100
       END                   . END SIND/CØSD
```

PROGRAM IDCU CODE: FIGURE 5 WS-1008-7-2

PROGRAMMER

DATE

```
. FØR BREVITY, WE DENØTE THE VARIABLES AS E1, E2, E6, EP0, EP1, EP2
. THE TWØ LARGE SUB-EXPRESSIØNS
.
.              MIN(EP0, LIM1)+(EP0-EP1)*E6
.          AND MIN(EP1, LIM1)+(EP1-EP2)*E6
.
. ARE COMPUTED BY THE SUBRØUTINE SUBEX
.
. THE FREQUENTLY USED TERMS E1, E2, EP0, EP1, EP2 ARE SLL MØVED TØ
. THE STACK AND LEFT THERE AT THE END

      BYT      0                 . ACCUMULATED LITERALS IN ØRDER ØF
                                 . ØCCURRANCE
                                 .
                                 , (FV0, FV1)
                                 .
                                 . DATA AREA REQUIRED FOR 9 VARIABLES
                                 .
                                 . EP0, EP1, EP2, E1, E2, E6, LIM1, LIM2
                                 .
                                 . ASSUMED TØ BE STORED IN THAT ØRDER,
                                 . PLUS RESULT DYPH
                                 .
                                 .
```

PROGRAM

PROGRAMMER   DATE

```
                                                                         BYTE  PTR
      SLL      E2,5        . TΦS=EP0, EP1, EP2, E1, E2                   000    5
      DΦ       2 BLT,U 1   .     =EP0, EP1, EP0, ETC.                      4    7
      LΦD,IH   (EV0,EV1)   . SET RETURN ADDRESS                            8    8
SUBEX XCH                  . SUB-EXPRESSIΦN: PTR REL ARG2                 11    2
      BLT,U    2           . TΦS=ARG2, ARG1, RA, ARG2                     12    3
      BLT,U    1           .     =ARG1, ARG2, ARG1, RA, ARG2              14    4
      SUB                  .     =ARG1-ARG2  (23.8)                       16    3
      SHA,L    23          .     =ARG1-ARG2  (0.31)                       17    3
      LΦD      E6          .     =E6(18.13)                               19    4
      SHA,L    13          .     =E6( 5.26)                               22    4
      MPR                  .     =E6*(ARG1-ARG2)  (6.25)                  24    3
      XCH                  .     =ARG1(23.8), PRΦDUCT(6.25)               25    3
      SHA,L    17          .     =ARG1(6.25)                              26    3
      BLT,U    0           .     =SAME TWICE                              28    4
      SKP                  .                                              30    4
      LΦD      LIM1        , TΦS=LIM1, ARG1  BΦTH  (6.25)                 31    5
      SUB                  .     =LIM1-ARG1  (6.25)                       34    4
      BRP      SXO         . BRANCH IF LIM1 > ARG1                        35    4
      PUP                  . LIM1 < ARG1                                  38    5
      BLT,D    2           . STΦRE LIM1 AT STACK 3                        39    5
      PDP                  . ADJUST PTR                                   41    4
SXO   PDP                  . TΦS=MIN(ARG1, LIM1)  (6.25)                  42    3
      ADD                  .     =MIN+PRΦDUCT  (6.25)                     43    2
      BLT,D    2           . SUBEX(ARG1, ARG2) TO STACK 0                 44    2
```

PROGRAM

PROGRAMMER DATE

```
                                                                          BYTE  PTR
      PDP                     . TΦS=RA, SUBEX(ARG1, ARG2)                  046   1
      IJP                     . RETURN FROM SUBEX CLOSE                     47   1
FV0   PDP                     . TΦS=SUBEX(0,1)                              48   6
      DΦ         2 BLT,U 3    . TΦS=EP1,EP2,SUBEX(0,1)                      49   8
      LΦD,RH     (FV0,FV1)    . SET RETURN ADDRESS                          53   9
      JMP        SUBEX        . EVALUATE SUBEX(1,2)                         56   9
FV1   PDP                     . TΦS=SUBEX(1,2),SUBEX(0,1)                   59   7
      SHA,L      6            .      =SUBEX(1,2)(0.31)                      60   7
      BLT,U      6            .      =E2(13.13),SUBEX(1,2)                  62   8
      SHA,L      13           .      =E2(5.26),SUBEX(1,2)                   64   8
      MPR                     .      =E2*SUBEX(1,2)(6.25)                   66   7
      ADD                     .      =PRΦDUCT+SUBEX(0,1)(6.25)              67   6
      BLT,U      0            .      =LAST THING TWICE                      68   7
      SKP                     . ITEMS NEXT LINE ALL (6.25)                  70   7
      LΦD        LIM2         . TΦS=LIM2,THING,THING                        71   8
      SUB                     .      =LIM2-THING                            74   7
      BRP        SV2          . BRANCH IF LIM2 > THING                      75   7
      PUP                     . LIM2 < THING                                78   8
      BLT,D      2            . STΦRE LIM2 AT STACK 6                       79   8
      PDP                     .                                             81   7
FV2   PPP                     . TΦS=MIN(THING,LIM2)(6.25)                   82   6
      BLT,U      4            .      =E2(13.13),MIN(6.25)                   83   7
      MPR                     .      =DYPM(5.6)                             85   6
      SKP                     .                                             86   6
```

PROGRAM

PROGRAMMER   DATE

```
                                                  BYTE
      STØ  DYP,I       , STORE RESULT             087  5
      ...                                          90
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

6.0 Additional Considerations

The process of developing the AGC architecture and programming representative G&N computations on the AGC suggests several desirable enhancements to the SPL/MK III language. These take the form of a series of "intrinsics" and other functions which are easily derived from the AGC instruction repertoire. By "intrinsic" we mean a computation which is written like a function call but which is performed with in-line code. There are several of these already in SPL/MK III and these are shown on page 2-69 of Part I of this report.

We repeat them here with the proposed new intrinsics together with an explanation of their function and how they may be coded in the AGC instruction set. A rationale for adoption is presented for the new ones.

6.1 Existing SPL/MK III Intrinsic Functions

6.1.1 .ABS This is applied as a unary operator to the quantity at the TOS by executing the operator ABS.

6.1.2 .REM Provides the integer remainder of the first argument divided by the second. If either argument is non-integer (exponent ≠ 0) it is first integerized by truncation. This function is accomplished by the coding

```
÷i      TØS = Remainder, Quotient
XCH     TØS = Quotient, Remainder
NIX     TØS = Remainder
```

6.1.3 .REMQUØ (arg 1, arg 2 = REM, QUO) Provides both the integer quotient (stored at QUO) and the integer remainder (stored at REM). Since this intrinsic is also accomplished by ÷i, the arguments are integerized if necessary by truncation. The coding is

```
÷i          TOS = Remainder, Quotient
STC REM           Remainder → REM
STC QUO           Quotient → QUO
```

6.2 Proposed Additional SPL/MK III Intrinsic Functions

6.2.1 .QUO To provide the integer quotient of two arguments. The rationale for adopting this function is first, that is already available thru the operation ÷i, but more importantly, the division symbol "/" stands unambiguously for single or double precision floating point division via the operators ÷R(S) or ÷R(D) hence there must exist a language form for specifying integer quotient. This problem does not arise in previous versions of SPL since they have been intended for fixed point computers. The coding is

```
÷i    TOS = Remainder, Quotient
NIX   TOS = Quotient
```

A less desirable coding (results not necessarily the same) is

```
÷R(S or D)  TOS = S or D flt pt. Quotient
INT         Integerize the Quotient
(TRC        if double precision)
```

6.2.2 .INT To integerize the single argument. The floating point structure of the AGC makes it unnecessary to provide a means for floating an integer, but the reverse is often a desirable capability. The coding is simply to apply the operator INT to the TOS. Thus, the second coding for .QUO, if desired can be gotten from .INT (arg 1/arg 2). (This form does not integerize the arguments prior to division.)

6.2.3 .FRAC To find the fractional part of the single argument. This capability has been found handy in such computations as quadrant reduction in trigonometric calculations. It is encoded by applying the FRAC operator to the quantity at the TOS. Assuming the .INT intrinsic, .FRAC can be coded without the FRAC operator but it is laborious:

```
DUP  )
INT  | 4 bytes (vs 1)
CHS  } 1 add (vs 0)
 +   )
```

6.2.4 .RND To allow rounding control over the results of intermediate results in a formula evaluation. Operating on rA, rounding takes place at the boundary between the left and right halves of that register. If APFF was on it is turned off. Thus .RND serves to "singularize" a double precision number by rounding. The coding is simply to apply the RND operator to the TOS.

6.2.5 .TRUNC To allow for singularizing a double precision intermediate result by truncation. This is encoded simply by using the TRC operator which turns off the APFF.

6.2.6 .SIGN To transfer the sign of its single argument to a 1. This traditional mathematical notation (also written SGN or signum) appears frequently in the definition of G&N equations in expressions such as

A = .SIGN (B) * C

Without the .SIGN intrinsic, the alternative SPL coding would be

$$\text{A} = \left\{\begin{matrix}-\text{C}\\ \text{C}\end{matrix}\right\}$$

$$\text{IF B} \left\{\begin{matrix}\text{GQ}\\ \text{LS}\end{matrix}\right\} \text{0 then A} = \left\{\begin{matrix}\text{C}\\ -\text{C}\end{matrix}\right\}$$

the alternative depending on the expected frequency of signature. This form requires 10 syllables, 16 bytes and 5 memory accesses. Somewhat better is

A = (1 - 2 * (B LS 0)) * C

recalling that B LS 0 = 0 (false) or 1 (true). This requires 10 syllables, 13 bytes, 3 memory accesses but also a subtract and two multiplies.

The .SIGN intrinsic is implemented with the SGN operator applied to the TOS. Accordingly, the above computation becomes

$$\left.\begin{array}{l}\text{FSS B}\\ \text{SGN}\\ \text{FSS C}\\ \ \ *\\ \text{STC A}\end{array}\right\} \quad \begin{array}{l}\text{5 syllables, 8 bytes, 3 memory accesses,}\\ \text{1 multiply}\end{array}$$

6.2.7 .MIN, .MAX These intrinsics operate on an unlimited number of arguments. The rationale for adoption is that it saves a great deal of SPL coding and when implemented via the MIN and MAX operators a great compaction in the object code is realized. These functions are quite frequent in their occurrance in G&N equations.

Consider for example

```
BIG = .MAX (A1, A2, ... An)
```

where the arguments are simple variables.

Without the .MAX intrinsic, this would be coded,

```
A1 = BIG

IF BIG LS A2 BIG = A2

IF BIG LS A3 BIG = A3
   .
   .
   .
IF BIG LS An BIG = An
```

This compiles to 6n-4 syllables, 10n-6 bytes and 3n-1 memory access ($n \geq 2$). With the .MAX intrinsic, using the MAX operator, the coding is

```
FSS A1  )
FSS A2  |
MAX     |
FSS A3  |  2n syllables, 3n + 1 bytes
MAX     >  n + 1 memory accesses (n ≥ 2)
  .     |
  .     |
  .     |
FSS An  |
MAX     )
```

A similar advantage is found for finding the extreme value in an array programmed with a FOR-loop.

MAX and MIN are binary operators which leave the maximum and minimum of their two operands on the TOS.

6.2.8 .DESCR This intrinsic has as its single argument an array identifier. The value returned is the array descriptor via an FST to the PCT or an FSC to the calling sequence. The chief use of making the descriptor available as data is that array arguments can be passed to subroutines without passing their dimensions as additional arguments. Thus, one might code

```
PROC .MXINV (A, EPSIL = ERR.) "Matrix inversion"
DECLARE (N, N) F7, A "Dummy dimensions"
    .
    .
    .
M = .DESCR (A) (7//5)     "Pick up first dimension"
N = .DESCR (A) (12//5)    "Pick up second dimension"
IF M NQ N RETURN (ERR)    "Abort if not square"
N = N + 1
    .
    .
    .
```

6.3 Proposed Additional Operators

6.3.1 LNØT, LQUIV These are not intrinsics but unary (LNØT) and binary (LQUIV) logical operators. LQUIV is provided since the EQUIV operator is already present to provide the Boolean operator of the same name. LNØT is provided via the CMP (32-bit ones complement) because ordinary mathematical operations and shifting operations are not available on the AGC to build logical masks via complementation.

6.3.2 XOR This is not an intrinsic but an additional Boolean operator which is provided since the XOR operator is already present to encode the SPL LXOR logical operator.

7.0 References

1. ADVANCED TARGETING STUDY PHASE 1A, Final Report, Vol. III, Part B, Equations and Related Analyses (U)

2. ADVANCED TARGETING STUDY PHASE 1B, Final Report, Vol. II, Computer Hardware Software Study (U)

3. SSTTP Universal Flight Program Document 12039-6142-R7-00 (U) 6 March 1970

4. GUIDANCE AND TARGETING EQUATION SPECIFICATION, CIRAD Report WS-1008-7-2, dated 2 June 1970.