To the Graduate Council:

I am submitting herewith a thesis written by Charles C. Ostrum entitled "An Intermediate Language Interpreter for the C-Based Mumps Programming Language." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

David W. Straight, Major Professor

We have read this thesis
and recommend its acceptance:

Jean R. S. Blair

Charles P. Pfleeger

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of Interlibrary Services when, in the opinion of either, the proposed use of the material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Charles C. Patrum

Date May 19, 1987

AN INTERMEDIATE LANGUAGE INTERPRETER FOR THE C-BASED MUMPS PROGRAMMING LANGUAGE

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Charles C. Ostrum

June 1987

ACKNOWLEDGEMENTS

The author wishes to express appreciation to Dr. Kevin C. O'Kane for his support and guidance throughout the author's course of graduate study and for providing the C-Based Mumps interpreter, without which, the completion of this thesis would have been impossible.

Additional thanks go to the thesis committee, Dr. David Straight, Dr. Charles Pfleeger, and Dr. Jean Blair for their support and interest in serving on the committee. I especially would like to thank Dr. Straight for assuming leadership of the committee in the absence of Dr. O'Kane.

The author would like to thank the management of the Naval Research Laboratory, USRD, for both their financial support and the use of their facilities in the conduct of this research.

Finally, the author would like to thank his father and mother for their financial support, encouragement, and faith throughout the authors academic career. The author is especially grateful to his wife, Debra, for her encouragement, support, and patience.

ABSTRACT

The C-Based Mumps programming language is an implementation of ANSI standard Mumps. It is a fully interpreted high-level programming language oriented towards string manipulation and database applications.

This project studied the potential for execution speed improvement by the elimination of parsing overhead while maintaining complete language functionality. This presents difficulties because the Mumps language definition includes dynamic code formation and execution which precludes full compiling.

Compilation difficulties are overcome by using an intermediate language. Mumps is translated into an intermediate language, which can then be interpreted without additional parsing. An intermediate language program is interpreted by a special interpreter that is integrated into the full Mumps interpreter. Dynamic situations are handled by communication between the two interpreters.

The major focus of the project was the design of the intermediate language and the implementation of its interpreter. In addition, an intermediate language translator was implemented by modifying a portion of the C-Based interpreter to generate intermediate code.

Project results show that the intermediate language technique is capable of maintaining complete Mumps language functionality. Execution speed improvements of from 3% to 45% percent (does not include translation time) were measured. The amount of improvement was directly related to the amount of underlying parsing involved. It was also found that Mumps execution time is heavily dominated by string manipulations and symbol table lookup.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis presents research into the potential for improving the execution speed of the C-Based Mumps programming language by the incorporation of an intermediate language interpreter into the C-Based Mumps interpreter. C-Based Mumps [1] is a fully interpreted implementation of the ANSI Standard Mumps programming language, written by Dr. Kevin C. O'Kane currently of the University of Alabama, Tuscaloosa, Alabama. It is written in C and runs primarily under Unix and Digital Equipment Corporation's (DEC) VAX/VMS operating systems. For this project, the intermediate language translator and interpreter are being written in DEC's VAX C language. The intermediate interpreter will be incorporated into a test version of C-Based Mumps running under VMS on a DEC MicroVAX II computer.

Interpreted Languages

Interpreted languages offer three principal benefits.

- They are convenient to use.
- They can include language constructs that by function cannot be implemented in a compiled language.
- They can be constructed so as to be easy to port to different machines and operating systems.

C-Based Mumps is not an exception to the above listed benefits. It presents the programmer with an interactive, conversational environment that makes both program development and database maintenance very convenient. C-Based Mumps includes numerous features, many of which are

extensions to standard Mumps, that are not found in other languages. Most of these features are non-standard in any language and are either impossible to compile or would be cumbersome and difficult to use in a compiled language environment. Finally, C-Based Mumps, by choice of implementation language and the avoidance of system calls, is readily portable to most interactive operating systems that have a C compiler.

All of the benefits of interpreting do not come without cost. It is a widely held conclusion that a program executed by an interpreter executes slower [3] than it would if it were compiled. This argument is persuasive for languages like Basic that do not contain features that preclude compiling. However, for interpreted languages such as Lisp, Prolog, and Mumps that contain either non-compilable features (Mumps) or semantics that do not readily adapt to the typical Von Neuman architecture (Lisp) [7], there can be no full comparison. It also can be argued that interpretation makes possible programs and development environments [7] that would not exist if we were limited to compiled languages such as Fortran, Cobol, and Pascal.

Regardless of one's stance on the interpreter versus compiler speed argument, it is worthwhile to explore improving interpreter execution speed. There are three theoretical reasons why compiled code is faster. First, interpreters must parse every source command before they execute it, and in the case of a program loop as many times as the loop is traversed. Compilers only parse the source program once. Second, full interpreters do not perform code optimization. Third, because they do not generate machine code, it is difficult for them to take advantage of the architectural features of the target machine.

Intermediate Languages

Intermediate languages have application to both the writing of faster interpreters and portable compilers [2,4]. The intermediate language, which is also interpreted, can be incorporated into the regular interpreter in such a way that all interpreter benefits are preserved. The programmer can still have full access to the convenience of the interpreter, all language features can be maintained, and portability can be preserved. Preserving these benefits while adding an intermediate language interpreter to the C-Based Mumps interpreter is a major design goal of this project.

To employ an intermediate language, the technique is to translate the source language into the intermediate language rather than machine language as would be done in a compiler. The intermediate language program is then interpreted by a virtual machine that produces the same program results as the regular interpreter or compiler does [2].

This technique produces the same potential for speed improvement that compiling does. First, all parsing is performed only once when the source program is translated into the intermediate form. Note, however, the intermediate language must be designed so that no parsing is required when it is interpreted by the virtual machine. Second, optimization techniques can be applied to the translated program. And finally, the virtual machine, because it is much smaller than the interpreter, can be constructed to take better architectural advantage of the host without unduly sacrificing portability. The latter advantage is pursued beyond a specific machine's instruction set and into microprogramming [3] and specialized interpretive architectures [7].

Intermediate Language Interpreter for C-Based Mumps

Three components are required to implement the intermediate language technique for C-Based Mumps. These are:

- The definition of the architecture and instruction set (the intermediate language) of the virtual machine which will be called the M-Machine.
- The production of a translator to translate Mumps into M-Machine code. In this project this is accomplished by modifying interpreter routines to produce code rather than execute code.
- The production of the actual M-Machine and incorporation into the C-Based Mumps interpreter.

In terms of focus the first component - the intermediate language and M-Machine architecture as related to C-Based Mumps - will receive the most attention. The Mumps compiler and the implementation of the M-Machine will be secondary. However, as was stated earlier, the implementation will not be a subset and will include all features and constructs of C-Based Mumps. The preservation of all language features will be considered a major design constraint. The optimization of the intermediate code, although possible, will not be addressed. optimization will not be addressed.

Finally, performance between the two implementations will be measured and compared. This will be done in such a manner as to identify the relative performance of specific operations so that suggestions for improvement can be made.

CHAPTER II

OVERVIEW OF HYBRID C-BASED MUMPS INTERPRETER

This chapter presents an overview of the features, behavior, and design of the hybrid C-Based Mumps interpreter. Included is a brief overview of the Mumps language and the C-Based Mumps superset in particular. The language overview is done so as to present the constraints and problems faced when designing an intermediate language for Mumps. This will be useful when the intermediate language is presented in later chapters. The reader should consult [1] for a complete description and specification of the C-Based Mumps language and interpreter implementation.

Mumps Language Overview

Mumps is a procedural language that is oriented towards database and data management applications. In support of this, it includes a built-in database facility and a comprehensive array of string manipulation and pattern matching operators. C-Based Mumps extends the built-in database into an hierarchical, tree structured facility. Also as an extension, there is a logic interpreter similar to that in Prolog, that uses the database facility for its fact and rule databases.

Besides its special features, Mumps also has the features found in more conventional programming languages such as Basic. These include program loops, conditional execution, absolute branches, subprograms, various forms of input/output, integer and real numeric calculation, and logical operations.

However, even in its more conventional features, Mumps is different. Where in conventional languages the program must be completely defined before interpretation or compilation, Mumps allows this to be done dynamically. Dynamic definition in Mumps is most notable in the areas of expressions, command arguments, the logic interpreter, and with the XECUTE command - Mumps code itself. The dynamics of Mumps, which precludes ever making Mumps a fully compiled language, greatly enhances the flexibility of Mumps.

Variables, symbol tables, and database facility. Mumps has three kinds of variables: local variables, local arrays, and global arrays. Both the global and local Mumps arrays are sparse. That is, only the defined elements exist. For example the existence of A(10) does not imply the existence of elements A(1) through A(10). Array subscripts may be either numeric or character. In either event, the array name and each instance of its subscripts denotes a unique name that identifies the array element. Except when the distinction is relevant, the term variable will be used for both variables and arrays.

Local variables are stored in the local symbol table and global variables are stored in the global symbol table. The local symbol table is contained in main memory while the global symbol table, which is the database facility, is disk resident. With some minor exceptions, local and global variables can be used interchangeably. Global variables require no special handling and are distinguished from local variables only by the first character of their name which must be a circumflex,

"^". However, global variables, because they comprise the database facility, are inherently more powerful.

The contents of both symbol tables are fully under program control at all times during program execution. Variables dynamically come into existence by being used as the target of a SET command (assignment) or a READ command. Local variables exist until deleted by the KILL command or the interpreter is terminated. Global variables exist until deleted by the KILL command or until the file that contains them is recreated or reformatted. In Mumps there are no declaration statements, nor are there explicit data types. C-Based Mumps maintains all data values as strings. The string will be automatically coerced into the data type implied by the operation being performed. For example, addition will force a numeric interpretation on the operands while the logical AND operator will force a logical interpretation.

The global variables form the basis of the built-in database. C-Based Mumps views each global variable as a path description in a tree structured hierarchy. Because a global variable is as easy to use in C-Based Mumps as a regular variable in conventional languages, complex database relationships can be manipulated with ease. No special database calls or access methods are required.

C-Based Mumps also uses the global variables to partition the database into a directory structure similar to UNIX directories. As part of this feature, there are built-in functions to navigate and use the directory structure. These include sub-directory creation, listings of both the sub-directories and their contents, and the specification of a sub-directory default.

Indirection. The most dynamic feature of Mumps is program indirection. Indirection takes place in Mumps when the actual code to be executed is contained in a variable rather than coded in the program. Mumps uses the "@" symbol (except in the XECUTE command which itself specifies indirection) to indicate that the contents of the variable or expression value is to be interpreted as code rather than data. Indirection, except in its simplest cases, absolutely prevents Mumps from being fully compiled.

The simplest form of indirection is one of the types of the DO (subprogram call) and GOTO (unconditional branch) commands. In this form, the indirect variable simply contains the file name containing the target code. This requires very little special handling and could be handled in conventional compiled languages by a dynamic linker/loader. Most operating systems do not support this except in the more trivial case of running a program. Typically all modules of a program must be defined when external references are resolved during linking. This is not sufficient for Mumps forms because the reference is not known until the program is executing.

Another simple form is one type of the KILL command. In this form, the indirect variable contains the list of variable names to be deleted. Because of its simplicity, the indirect string can readily be handled by the intermediate language interpreter and presents no major problem.

One of the two complex cases of indirection is expressions. In this form the indirect variable contains either a partial or complete expression. This requires a complete infix to postfix translator and a postfix evaluation module to interpret the expression.

The most complex case is the XECUTE command. In this form, the evaluation of the indirect variable or expression results in a complete line of Mumps code that must be interpreted. The code may contain any command including recursive XECUTEs. This form requires all parts of the C-Based Mumps interpreter in order to execute the command.

Logic interpreter. The logic interpreter of C-Based Mumps contains many of the features found in the Prolog language. Mumps predicates are global variables which can be either rules or facts. The sub-directory system may be used to specify different sub-directories for the rules and facts. Thus, rule-based logic systems that make use of existing data collection programs and databases can readily be implemented. Because the logic system has at its disposal the complete Mumps language, operations that are difficult to perform with the more traditional logic languages can easily be performed in C-Based Mumps.

Logical rules in C-Based Mumps contain a fourth kind of variable, called a predicate variable. A predicate variable is actually a character string global array subscript that begins with the "~" character. The logic interpreter stores all predicate variables in the local symbol table during rule evaluation. If the evaluation fails, the logic interpreter deletes the predicate variables. If the evaluation succeeds, all intermediate predicate variables are deleted, and the results are stored in local variables whose names correspond to the predicate variables embedded in the rule except that the "~" is removed.

A rule has two parts. The head is the conclusion to be drawn and the tail is a list of conditions that must be satisfied for the rule to

evaluate true. The head is the global array specification and the tail is the data stored in the global array element. Rule tails may contain Mumps expressions that reference all four types of Mumps variables and all built-in functions and variables. Assignment of values is permitted and all assignments are permanent, if made to other than predicate variables. Indirection is also permitted.

The logic interpreter and the predicate variables can be used to create a relational view of the global array database. Relational operations that are possible include selection, projection, join, intersection, minus, and union.

Mumps Programs and Programming Environment

Mumps has two modes of operation: direct or interactive mode and program mode. All features of the language are available in both modes. The only distinction between the two is the number of command lines executed before returning to the interactive user or command file (VMS, UNIX, etc.).

In direct mode, which is active upon starting the interpreter, the user receives the prompt signifying that Mumps is ready to accept a command line. The user then enters one or more commands. Mumps executes the commands and prompts for the next command line. Program mode is entered from direct mode via the DO command. When Mumps completes the program specified in the DO command, it returns to direct mode. For this research, direct mode has no importance other than as entry into program mode and will not be discussed further.

In keeping with its conversational, dynamic nature, Mumps has no formal definition of what constitutes a Mumps program. Therefore, there are no explicit main programs, subroutines, or functions (in the sense of Fortran or PL/I functions). Instead a Mumps program or subprogram is a sequential list of one or more lines of Mumps commands that are grouped into standard text file we will call a program file. There are no formal argument lists, and the subprogram inherits the complete Mumps environment.

A single program file may be a complete program with or without subprograms, a subprogram or subprograms, or simply a program fragment. Logical completeness is determined solely by the programmer not the language. Behavior, however, is specified. The program is invoked by the DO command and begins execution either at the first line or at an alternative entry point specified in the DO command. Upon termination, control will return to the command immediately following the invoking DO, if the command exists; otherwise direct mode is resumed. During execution, the program file has access to all symbol tables and internal condition codes. If the internal condition codes are modified, then the modifications persist after return. During execution, the program file may invoke other program files including itself.

The C-Based Mumps Interpreter

The C-Based Mumps interpreter is completely written in the C language and is organized in a modular fashion. The interrelationship of the major modules is shown in Figure II-1. The function of each module is briefly explained in the following sections.

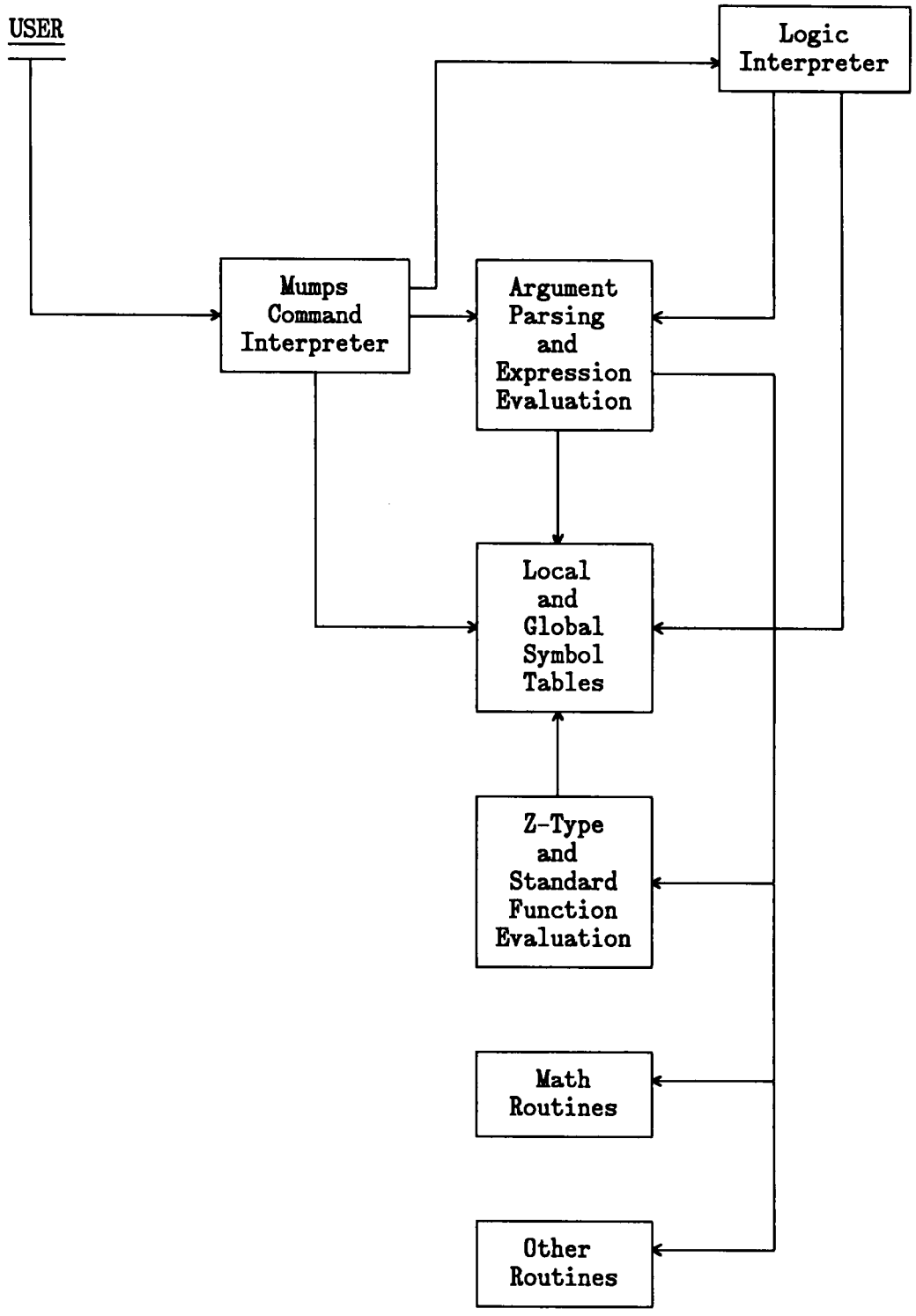

Figure II-1. Structure of C-Based Mumps Interpreter.

Mumps command interpreter. This is the interpreter's main program. It contains the initialization code, manages all control structures, and interprets all Mumps commands and extended program mode commands. There are three major control structures: an input/output control block, a stack for controlling FOR loops, and a stack for controlling DO and XECUTE commands. The code is organized around one major loop. Each transit of the loop results in one command being identified and executed. The main loop has special handling for errors, end of line, and end of program.

Logic interpreter. The logic interpreter is implemented as a separate interpreter in its own module. It is invoked through an added program mode command (ZP) in the Mumps interpreter. Communication with the main interpreter is through the local symbol table and the built-in variable, $Test. Each invocation results in one rule being evaluated. Control does not return to the command interpreter until evaluation is complete.

Argument parsing and expression evaluation. The parser is used by the command interpreter to do the major portion of the evaluation of all command arguments including embedded expressions. An infix to postfix recursive descent algorithm is employed to evaluate expressions. Because it performs evaluation as well as parsing, the parser accesses both symbol tables to evaluate variables, recognizes all built-in variables and functions, and handles all operators. As an adjunct to the logic

interpreter, the parser evaluates all Mumps expressions embedded in the rules manipulated by the logic interpreter.

Local and global symbol tables. Although these are grouped as one module, they are in fact implemented as separate modules. They are combined here because they are functionally similar. The symbol tables are both accessed through interfaces that hide the underlying data structures from the rest of the interpreter. The local symbol table interface has five functions: create/store, look up the current value, delete specific variables, delete the table, and find the next higher subscript of an array. The capabilities of the global symbol table interface are similar.

Z-type and standard function evaluation. There are two function evaluation modules. One for the standard Mumps functions and variables and one for C-Based Mumps extensions which are called Z-Type. Additional Z-type functions can be added by simply including the functions' code in the Z-type module and rebuilding the interpreter. No other routines need be changed.

Math routines. All mathematical operators are implemented as subroutines and are collected into one module. Their subroutine interfaces enable their implementation to be changed independent of the rest of the interpreter.

Other routines. The interpreter employs many internal service routines and miscellaneous routines for special operations. These are collected into the service module.

Hybrid Mumps Interpreter

The hybrid interpreter is basically the regular C-Based interpreter with the addition of the M-Machine module. For all Mumps code except indirect expressions, XECUTE commands, and logic interpreter calls (ZP), the M-Machine, when executing intermediate code, replaces the command interpreter and command argument parser. All other modules including the symbol tables, built-in functions and variables, math routines, and other routines are used by the M-Machine to interpret the intermediate code. The structure of the hybrid interpreter is illustrated in Figure II-2.

The M-Machine is started into execution by the ZXtended added program mode command which may be issued from either direct or program mode. In behavior, the ZX command is similar to a DO command. It has one argument that specifies a file containing intermediate code to execute. This file is loaded into the program memory of the M-Machine and execution begins. Upon program termination (HALT, QUIT, or program end), control is returned to the regular interpreter. Thus the M-Machine may be used to execute either complete or partial programs.

This basic design of the hybrid interpreter accomplishes two important objectives. First, because the M-Machine is incorporated into the regular interpreter, it has the full services of the interpreter at its disposal. This enables the M-Machine to handle the more complex

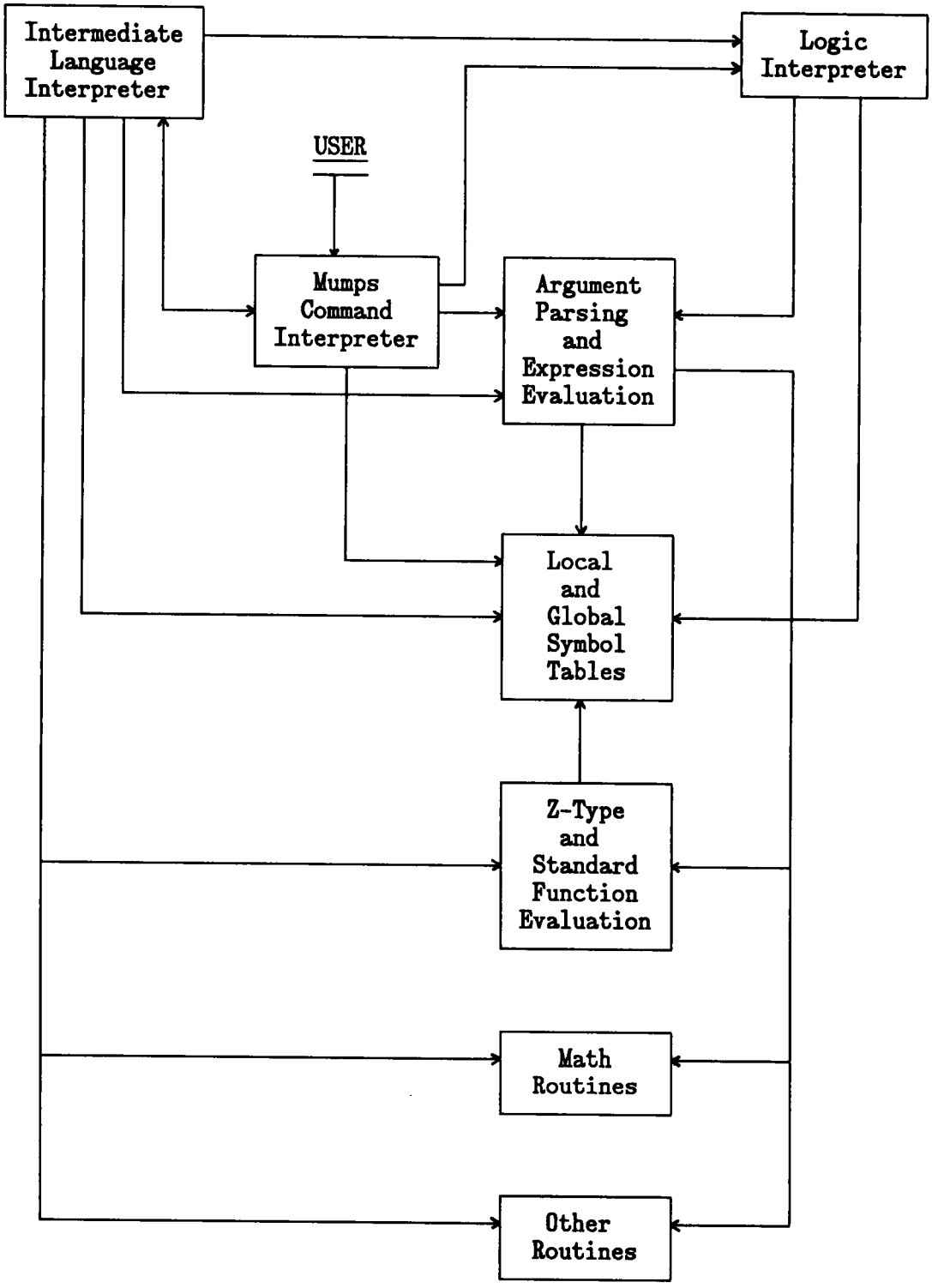

Figure II-2. Structure of C-Based Mumps with added Intermediate Language Interpreter.

cases of indirection (expressions, XECUTE commands, and calls to the logic interpreter) which cannot be handled in the intermediate language. The communication between the M-Machine and the main interpreter will be bidirectional and will be controlled by a shared data structure similar to the one employed in the main interpreter to control DO and XECUTE commands.

Second, execution of intermediate code, other than the method of invocation, is transparent. After execution, it is impossible to tell whether a program was executed by the regular interpreter or the M-Machine. This is accomplished by sharing certain data structures between the regular interpreter and the M-Machine. These are input/output, all built-in variables - particularly $Test, and the symbol tables.

Intermediate Language Translator

An intermediate language translator is provided to translate Mumps program files into intermediate language program files. It was constructed by modifying the main interpreter and parsing modules to generate intermediate code rather than interpret Mumps code. Each program file that is translated produces one intermediate program file. Like Mumps program files, there are no restrictions except that the generated code must not exceed the available memory space of the M-Machine. The generated code requires no relocation and can be directly loaded into the M-Machine. No provision is made for independent translation and the subsequent linking of modules into programs. However, as will be seen in later chapters, there is nothing in the structure of an intermediate program that would prevent linkage editing.

CHAPTER III

ARCHITECTURE OF INTERMEDIATE LANGUAGE INTERPRETER

The intermediate language interpreter, the M-Machine, is a virtual machine that is implemented in software. In operation, it behaves similarly to an ordinary physical computer. A program is loaded into its memory space and the program counter (PC) is loaded with the address of the first instruction to execute. When this instruction is completed, the PC is updated with the address of the next instruction. Execution continues until either the program ends or an error occurs or a program termination instruction is executed.

The implementation of this basic architecture is straight forward. It is a program loop that contains a case statement. Each case is the operation code of an M-Machine instruction. Each transit of the instruction cycle loop results in one instruction being executed. Figure III-1 illustrates the flow of program execution.

Stack Architecture

A stack based architecture was chosen for the M-Machine because of its simplicity and power [2,3,4,5,6]. In addition, a stack structure for the M-Machine parallels the internal structure of the C-based Mumps interpreter. As an added benefit, the size of the object code is smaller for stack machines than for register machines [2].

The M-Machine employs three stacks: one general purpose (expression) stack and two control stacks. The general purpose stack is used in the evaluation of postfix expressions [2,3,4,5] (all expression

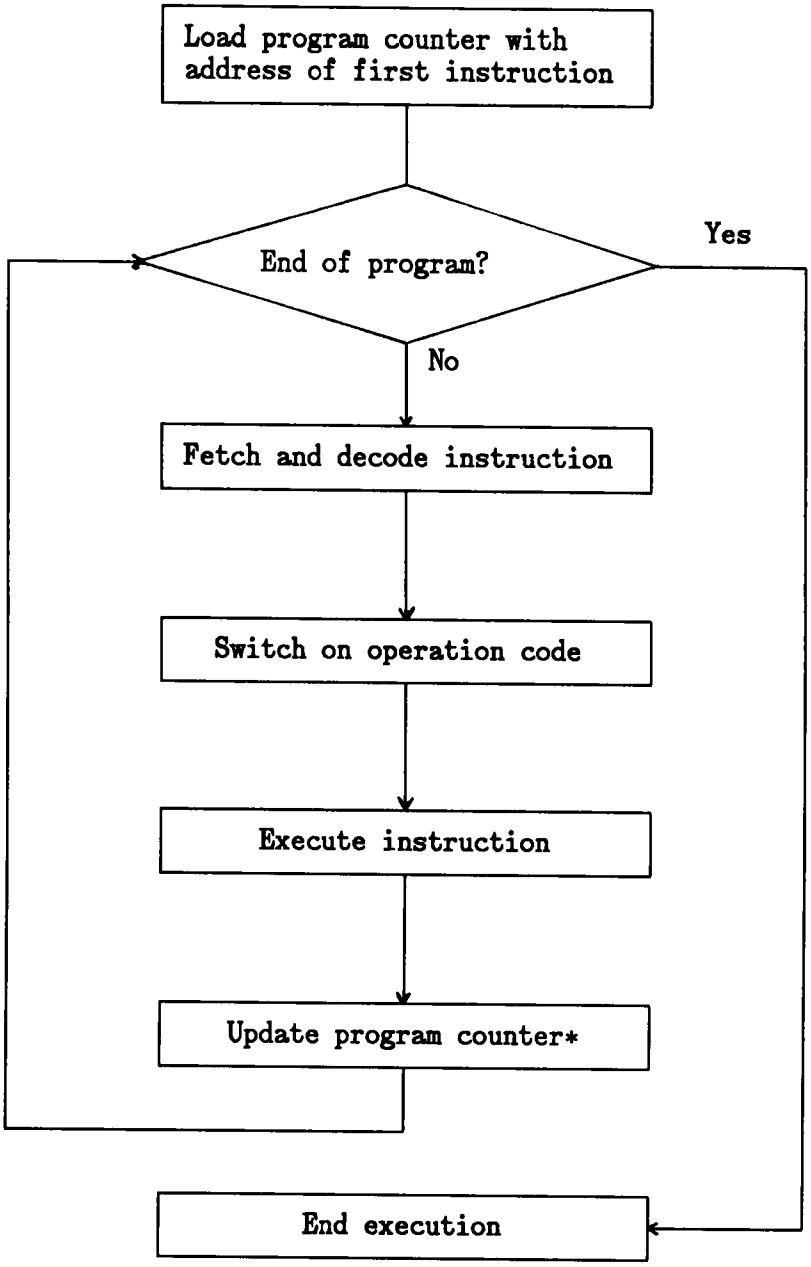

Figure III-1. Flow of M-Machine Program Execution.

*In the case of instructions that result in program branches, the program counter is updated in the course of instruction execution and this step is skipped.

code sequences generated by the translator are in postfix form) and as temporary storage for passing operands and control information between instructions.

One control stack is for loops and the other is for subprograms. Both control stacks have similar counterparts in the C-based interpreter. The stack data structure has the ability to handle program nesting. As each nesting level is entered, the control information for that level is pushed onto the stack. When the level is exited, the control information is removed from the stack [2]. The containing level, if there is one, then resumes control. More detailed discussion of these stacks is deferred until the next chapter when the execution of Mumps FOR, DO, GOTO, and XECUTE commands is presented in detail.

Instruction Set

The M-Machine employs a high-level instruction set. A high-level instruction is typically more complex and performs more operations than a low level instruction. The principal benefit of this is a reduction in the space required by the object code [2]. Reduced size object code was considered desirable because of the historical use of MUMPS on small machines that have a program memory space of 64K bytes.

Two instruction formats are used: no operand which requires two bytes of storage, and single operand which requires four bytes. In both types the first byte is the instruction length and the second is the operation code. In the single operand instruction bytes three and four contain M-Machine addresses. The instruction formats are illustrated in Figure III-2.

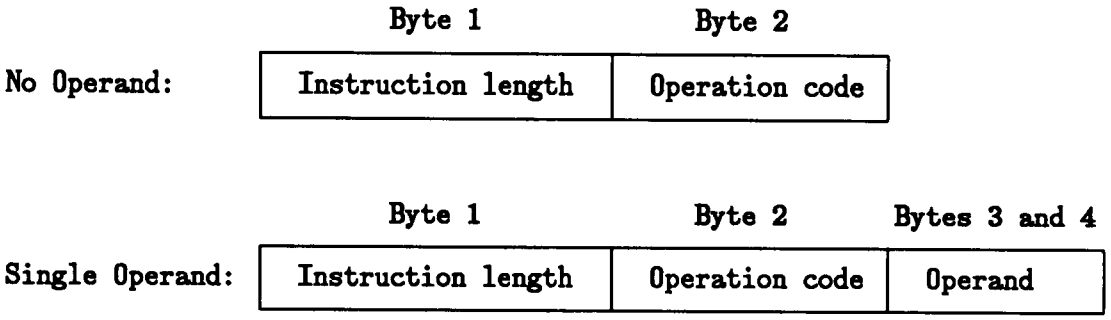

Figure III-2. Instruction Formats.

Additional instruction formats are not necessary because of the stack architecture. Many instructions, particularly the unary and binary operator instructions, such as ADD, implicitly address their operands by accessing either the top or top two operands on the stack.

The operand in the single operand instruction contains either a branch address or addresses a constant or variable name within the intermediate module. The operand is never an immediate value (contained within the instruction) because all C-based MUMPS constants are character strings requiring variable length storage. All constant strings are collected into special memory sections rather than being part of individual instructions. As an added benefit this eliminates storing multiple copies of all constants.

Memory Organization

The MUMPS compiler produces a module that is loaded directly, without relocation, into the program memory array of the M-Machine starting at location zero. All addresses (actually array indices) within the module are absolute (relative to location 0). The module is divided

into five sections as illustrated in figure III-3. Because all addresses are two byte integers, alignment of all instructions and entries within each section is MOD two in order to accommodate any alignment restrictions imposed by the underlying physical machine.

Map section. This section maps the layout of the module. All addresses locate the first byte of the respective section. They also serve as ending locations for the previous sections. For example, the M-Machine can determine program end by testing that the updated PC is less than the starting address of the label section. If the test fails, the end has been reached.

Code section. This section contains all executable code. All embedded addresses are either branch targets or references to entries in the constant or variable sections. Internal references are never made to a label in the label section. This is because label addresses are computed during compilation and are already coded in the instruction.

Label section. This section contains all labels present in the source program file. These are global symbols and represent potential entry points of the module (although they may not be used as such). This section is searched when a program is loaded and the invoking DO or GOTO command specified both a label and a file name. Each label entry consists of the length of the label entry, the label's value, and the null terminated label string. The value represents the program address of the label in the code section.

LOCATION	CONTENTS
0:	Address of code section
2:	Address of label section
4:	Address of constant section
6:	Address of program variable section
8:	Address of heap storage
Contents of 0:	Code section
Contents of 2:	Label section (global symbols)
Contents of 4:	Constant section
Contents of 6:	Program variable section
Contents of 8:	Heap storage
Memory size - 2:	Base of general purpose stack
Memory size - 1:	End of memory

Figure III-3. M-Machine Memory Map.

Constant section. This section contains all text and numeric constant strings generated by the compiler. Each entry consist of a two byte length followed by a null terminated string.

Variable section. This section contains the names of all program variables, both local and global, used in the module. The variables are sorted by type with the global variables being first. Entries in this section are identical to those in the constant section. The separation of global and local variable names was done in anticipation of adding a cache symbol table for the local variables.

Memory Management

Like the C-Based interpreter, the M-Machine is capable of executing a program whose total memory requirement is larger than available program memory. This is accomplished by program swapping and replacement. Swapping occurs when a DO command that specifies a different program file, is executed. This results in the currently loaded program being replaced with the specified program. However, unlike normal swapping, the replaced program is not written to a swap file. This is because, at the time of the swap, there is no dynamic information within the program memory. When the target program completes, the swapped program can be resumed by reloading its program file. Execution is then resumed using information that was pushed onto the control stack when the original swap occurred.

Program replacement occurs when a GOTO command that specifies a different program file is executed. Unlike, the DO command, the GOTO is

an absolute transfer without a return path. In this case, the currently loaded program is replaced with the specified program. All existing return information is discarded and the new program becomes the new base program. More detailed descriptions of these operations are presented in Chapter IV when the DO and GOTO commands are discussed.

Memory utilization. Because the M-Machine has only minimal memory management capabilities, efficient memory utilization and the consequent reduction of swapping is the responsibility of the programmer. As a rule, program files should be as large as possible and all frequently used subprograms should be in the same program file as the caller. Besides swapping, large program files usually require less total memory than a collection of small files. This is due to the fact that all constant strings and variable names are only stored once in each compiled module.

The ease with which programmer controlled memory management is accomplished could be improved by providing a linkage editor to convert independently compiled modules into larger programs. Although not a part of this research, a linkage editor for Mumps intermediate language modules is possible.

CHAPTER IV

MUMPS INTERMEDIATE LANGUAGE

The design for the Mumps intermediate language is based on the decomposition of the Mumps language into the individual actions required to execute all Mumps programs. Each action is then transformed into an intermediate language instruction that performs the action. Certain instructions are specific to individual commands or features of Mumps, while others are more generalized and are used by numerous commands and features. The former situation is exemplified by the FOR command which can require more than ten specialized instructions such as increment the loop control variable or perform a limit test. These instructions are specific to the FOR and are not used to execute other Mumps commands. The latter situation is exemplified by expression evaluation which requires instructions for operand fetches and operator application. These instructions are used throughout Mumps whenever an expression must be evaluated in the performance of a command.

The intermediate language presentation is made in the context of the C-Based Mumps language. First, the commands and features of Mumps are functionally grouped. Then, within each functional group, the individual commands or features are presented. The presentation includes a brief description of the command or feature, its specific intermediate instructions, and any control structures within the M-Machine used by the intermediate instructions. Because the presentation in Chapter IV is more concerned with the design of the intermediate instructions, the reader is referred to Appendix A for precise instruction definitions.

Expression Evaluation

Expression evaluation occurs as a sub-part of numerous Mumps commands. These include SET, post conditional, IF, ELSE, FOR, etc. However, regardless of where it occurs, it is uniformly handled. The expression may be either numeric, string, or logical, and in the trivial case, it may be a constant. Regardless of type, all operands are in string form and are implicitly data typed by the operation being performed. For example, the ADD operation will force a numeric interpretation on the operand strings regardless of whether they are numeric or not.

The translator translates all source expressions, which are in infix, into a postfix sequence of operand fetches and operator applications. There are numerous operand sources: constants, local variables, local arrays, global arrays, built-in functions and variables, indirect expressions, and operator produced intermediate results. For the first two sources, constants and local variables, there is a corresponding intermediate instruction, C_FETCH_CONSTANT and C_FETCH_LOCAL, that locates the operand directly and pushes its value onto the stack. The other sources are more complex and are discussed in the following sections. However, regardless of type or complexity, each operand fetch results in one operand being pushed onto the stack.

Operators and intermediate results. Each Mumps operator has a corresponding intermediate instruction. All operator instructions locate their operands on the general purpose stack. Unary operator instructions use the top operand, while binary operator instructions use the top two

operands. In operations that are not commutative (subtraction, division, etc.), the deeper operand is the leftmost in the infix representation. For example, the subtraction operation uses the top operand as the subtrahend and the top-1 operand as the minuend. When the operator instruction executes, it pops the required number of operands from the stack, performs the operation, and pushes the result back onto the stack. When the expression evaluation is complete, one value, the final result will be on the top of the stack. An arithmetic expression evaluation instruction sequence is illustrated in Figure IV-1.

In the current implementation, all operator instructions use the C_Based interpreter's evaluation routines. For example, the C_ADD instruction calls the subroutine, add(), in the math module of C-Based Mumps.

Arrays. Array subscript specifications are recursive. That is, each subscript is also an expression that must be independently evaluated. The expression can be any valid Mumps expression and can contain additional array references. After all subscripts are evaluated, they must be collected and combined with the array name to produce the complete array specification. The specification is then used to obtain the operand from the appropriate symbol table.

Each array specification evaluation begins by partitioning the stack into a substack. This is accomplished by the instruction, C_STACK_MARK which pushes a special code (M_STACK_MARK) onto the stack. Each subscript expression is then evaluated and the final result left on the stack. When all subscript expressions have been evaluated, all values

Command	Operand	Comment
C_FETCH_CONSTANT	10	Push the address of the operand string onto the general purpose stack.
C_FETCH_LOCAL	c	Use the variable name operand, "c", to fetch value from local symbol table. The value is copied into heap storage and the address pushed on the stack.
C_FETCH_CONSTANT	12	Push address of "12" onto stack.
C_SUBTRACT		Pop "12" and value of "c" from stack and perform "c" - "12". The result is copied to heap and the address pushed.
C_ADD		Pop twice, add, and push result.
C_FETCH_LOCAL	a	Fetch value of "a" and push.
C_FETCH_LOCAL	b	Fetch value of "b" and push.
C_DIVIDE		Pop twice, divide, and push result.
C_MULTIPLY		Pop twice, multiply, and push result.
C_STORE_LOCAL	k	Pop top value from stack and use variable name operand, "k", to store value in local symbol table. The stack is now empty.

Figure IV-1. The Intermediate Instruction Sequence to Perform: set k=(10+(c-12))*(a/b).

between the stack top and the stack mark are subscript values. The top value is the rightmost subscript and the value on top of the mark is the leftmost. Depending on whether the array is local or global, either the C_FETCH_LOCAL_ARRAY or C_FETCH_GLOBAL instruction is used to build the complete specification, lookup its value via the appropriate symbol table, and push the result onto the stack. When the specification is built, the stack is popped back to the stack mark. After the fetch is performed, the mark is replaced by the result.

The above technique correctly handles nested array specifications. Each fetch instruction simply obtains its subscripts by popping the stack back to the nearest stack mark. That mark is then replaced by the lookup value and becomes an operand in the overall expression or a subscript in another array specification. Figure IV-2 illustrates the instruction sequence for a moderately complex array reference, ^g1(a(b((c-1)+2,1)),"def"). In this specification, a total of 3 array specifications must be evaluated. The translator recognizes an array specification by the occurrence of the first open parenthesis immediately following the array name and generates a C_STACK_MARK instruction. Translation then continues to the deepest nesting level for the first subscript of ^g1. At this point the arithmetic expression, (c-1)+2, for the first subscript of b is encountered. The translator recognizes that the open parenthesis preceding "c" is for expression precedence and does not generate a stack mark in response to it. The close parenthesis is handled in a similar fashion. In the translator's internal stack, if the close parenthesis matches an expression precedence open parenthesis, it is treated as a precedence

```
    Command                 Operand     Comment

    C_STACK_MARK                        Mark for "^g1" specification.

    C_STACK_MARK                        Mark for "a" specification.

    C_STACK_MARK                        Mark for "b" specification.

    C_FETCH_LOCAL           c           Fetch "c" and push.

    C_FETCH_CONSTANT        1           Fetch constant "1" and push.

    C_SUBSTRACT                         Pop twice, subtract, push.

    C_FETCH_CONSTANT        2           Fetch constant "2" and push.

    C_ADD                               Pop twice, add, push.

Stack: mark, mark, mark, first "b" subscript.

    C_FETCH_CONSTANT        1           Fetch constant "1" and push.

Stack: mark, mark, mark, first "b" subscript, second "b" subscript.

    C_FETCH_LOCAL_ARRAY     b           Pop the two subscripts of "b",
                                        form specification, lookup, and
                                        replace mark with result.

Stack: mark, mark, first "a" subscript.

    C_FETCH_LOCAL_ARRAY     a           Form "a" specification, lookup,
                                        and replace mark with result.

Stack: mark, first "^g1" subscript.

    C_FETCH_CONSTANT        "def"       Fetch constant "def" and push.

Stack: mark, first "^g1" subscript, second "^g1" subscript.

    C_FETCH_GLOBAL                      Form "^g1" specification, lookup,
                                        and replace mark with result.

Stack: value of "^g1" array element.
```

Figure IV-2. The Intermediate Instruction Sequence to Evaluate the Array Specification: ^g1(a(b((c-1)+2,1)),"def").

close parenthesis. Otherwise, it indicates that an array or function reference has been completed, and the appropriate fetch instruction is generated.

Functions. C-Based Mumps has two types of built-in functions: the standard functions defined in the Mumps standard and the Z-type, which are defined by the author. The only difference between the two in C-Based Mumps is the evaluation module. The module, fcn() evaluates most standard functions except for $DATA, $NEXT, and $ORDER. These latter three functions, which deal with variables and arrays, are evaluated directly in conjunction with the appropriate symbol table manager. The built-in variables, which are similar to functions except for the lack of arguments, are also evaluated by fcn(). The added Z-type functions are evaluated by the module, zfcn().

Each standard function, built-in variable, and Z-type function has a corresponding M-Machine instruction. The reader is referred to Appendix A for a complete list of function instructions. Function evaluation is similar to array evaluation except instead of subscripts, there are arguments. Like subscripts, each function argument can be any valid mumps expression including array and function references. At the beginning of evaluation, the stack is marked with the C_STACK_MARK instruction. The arguments are then evaluated in successive order, with each argument expression evaluation leaving the final result on the stack. Next the specific function instruction builds the function specification, pops the arguments from the stack, calls the appropriate evaluation module, and replaces the stack mark with the result.

There are two exceptions: $PIECE, used on the left hand side of an assignment expression, and $SELECT. The psuedovariable use of $PIECE, which is similar to the psuedovariable use of the SUBSTR function in PL/I, is covered in a later section. $SELECT differs from the other functions in that each argument is a pair of expressions separated by a colon. Each sub-argument expression is evaluated separately and the result stacked. The C_$SELECT instruction then processes the stack contents as argument pairs rather than as individual arguments as do the other function instructions.

Before leaving arrays and functions, a note on the internal handling of array and function specifications is appropriate. C-Based Mumps array specifications are keys into either the local or global symbol table data structures. Because they are keys and the collating order must be maintained, the three separator characters, "(", ",", and ")" are translated into special 8 bit codes above the standard ASCII seven bit codes. Doing so both preserves the collating order and eases recognition during translation. Functions also use these codes as separators, along with a code for the ":" character in $SELECT compound arguments. When the M-Machine builds the array or function specification, the appropriate special codes are inserted between specification elements.

Indirect expressions. Mumps expressions and sub-expressions can be indirectly specified by storing the expression or sub-expression in a variable. Expressions of this type, because they are dynamically formed, cannot be directly evaluated by the M-Machine.

Indirect expressions are handled in the intermediate language by first using one of the three fetch code sequences discussed previously to obtain the contents of the indirect target variable. Next, the C_INDIRECT instruction is used to evaluate the indirect code. C_INDIRECT first pops the indirect code from the stack. Next, the interpreter module, parse_(), is called to evaluate the expression. Finally the evaluation result is pushed on the stack. The only restriction on sub-expressions is that each must correctly evaluate to a single result. An indirect sub-expression behaves as if it were parenthesized within the main expression.

Assignment

There are three commands or functions that can be used to assign a value to a variable: the SET command, the left hand side use of $PIECE function mentioned previously, and the FOR command. The setting of the loop control variable in the FOR command is not substantially different from the SET command and is presented in the FOR command section. In addition, the logic interpreter, transparent to the M-Machine, creates and modifies variables during its execution. All variables created or modified by the logic interpreter (excluding temporaries for intermediate results) are available for use by the M-Machine.

The SET command is accomplished by first evaluating an expression, then storing the result into either a global or local variable. Variable storage, depending on variable type, is handled by one of the three available instructions: C_STORE_LOCAL, C_STORE_LOCAL_ARRAY, and C_STORE_GLOBAL. The C_STORE_LOCAL instruction simply pops the result of

the right hand side expression evaluation from the stack and stores the value by calling the local symbol table manager. The instruction sequence in Figure IV-1 illustrates a SET command and the C_STORE_LOCAL instruction. The array storage instructions operate similarly to the array fetch instructions except that the stack top is the value to be stored rather than the rightmost subscript. First the value is popped, then the array specification built, the subscript values popped, and the stack mark popped. Finally the appropriate symbol table manager is called and the value stored. Upon completion of all store operations the stack is empty.

The left hand side use of the $PIECE functions is substantially different from the right hand side use. In the right hand side use, $PIECE locates, extracts, and returns a specified substring from the target expression. In the left hand side use, the target expression must be contained in a variable (local or global). The specified substring is located within the variable's value and is replaced by the value of the right hand side expression. After replacing the sub-string with the right hand side value, the variable is updated in the symbol table.

The instruction, C_$PIECE_LHS, is used to perform all of the left hand side operations except for updating the variable. This instruction pops the right hand side result from the stack, builds the function specification, pops the stack back to the stack mark, calls the fcn() module to locate the substring, replaces the substring with the right hand side value, and replaces the stack mark with the result. The instruction used next depends on the target variable type. If it is a local variable, C_STORE_LOCAL is used. If it is an array, either

C_$PIECE_LHS_SLA for local arrays or C_$PIECE_LHS_SGA for global arrays is used. The $PIECE array storage instructions differ from the standard array storage instructions in that the value to be stored is on top of the stack mark prior to the subscript values rather than on top of the stack after the subscript values. Otherwise, the operation of both types of array storage instructions is identical. This inverted situation for arrays results from translating the storage sequence after the $PIECE left hand side evaluation sequence.

Symbol Table Management

As described in Chapter 2, creating names in both the local and global symbol tables is dynamic and requires no pre-allocation or specification of variable names and data types. A name is added into the appropriate symbol table and defined simply by using the name in a command that results in a value being stored into the name. In the C-Based interpreter this action is handled for the local symbol table by the routine, sym_(). Whenever sym_() is invoked and a store is specified, sym_() will first search for the name. If the name is found the value is updated. If the name is not found, it is added to the symbol table and the value stored. The same results are accomplished for global variables by the global routine, global().

To conserve space, variables, including arrays, can be deleted from both the local and global symbol tables with the KILL command. It has three forms: delete the complete local symbol table, remove a specified local or global variable, and from the local table only remove all variables except for those specified. For the second and third forms,

target variables can be indirectly specified. That is, the target variable name is contained in the specified variable. For example, if k="a(1,2)", the command, "kill @k", results in array element, "a(1,2)" and its descendants being deleted. For global variables, in addition to the KILL command, the complete global symbol table can be initialized, destroying the current contents, with the ZG added program mode command. The ZG command is discussed in a later section.

Because of its many forms, the KILL command requires numerous intermediate instructions. The first form, delete the complete local symbol table, is performed by the C_KILL instruction. This instruction simply sets the local symbol table extent to zero.

The second form, delete a specified variable, is handled by four specialized instructions. Three instructions, one for each variable type, are for directly specified targets. The instructions are C_KILL_LOCAL, C_KILL_LOCAL_ARRAY, and C_KILL_GLOBAL. All three instructions are single operand, with the operand being the name of the variable to delete. C_KILL_LOCAL simply calls sym_() with the operand variable name and delete specified. The two array instructions first build the complete array specification as previously described, then call either sym_() or global() to affect the deletion.

For indirect specifications, there is one specialized instruction, C_KILL_INDIRECT. In addition, the appropriate fetch code sequence is used to obtain the contents of the indirect variable. C_KILL_INDIRECT, like C_INDIRECT, pops the indirect code from the stack and calls the interpreter routine parse_() to evaluate the specification. The result returned from parse_() is the variable to delete. The appropriate symbol

table manager is then called to affect the deletion. The call to parse_() in the C_KILL_INDIRECT instruction is different from the call in C_INDIRECT. In C_KILL_INDIRECT, the specification is evaluated as if it were on the left hand side of an assignment statement, while in C_INDIRECT, the expression is evaluated as the right hand side of an assignment statement. For example, take the specification ^g1(a(1)) where a(1)=4 and ^g1(a(1))="abc". The call to parse_() in C_KILL_INDIRECT will return ^g1(4) in internal form, while in C_INDIRECT, the call will return "abc", the contents of ^g1(a(1)). The third form, delete all but the specified variables, applies only to the local symbol table. This form is implemented with five specialized intermediate instructions. Two instructions, C_KILL_LOCAL_EX and C_KILL_LOCAL_ARRAY_EX, apply to directly specified targets. The two analogous instructions for indirectly specified targets are C_KILL_AT_LOCAL_EX and C_KILL_AT_LOCAL_ARRAY_EX. These four instructions operate similarly to the second form instructions except, rather than deleting the target, the target specification is pushed on the stack.

The algorithm used to implement this KILL form in the interpreter is to build a list of all variables to retain. After the keep list is built, each variable in the local symbol table is inspected, if it is not on the keep list, it is deleted. In the intermediate language, the translator first generates a C_STACK_MARK instruction when this form is encountered. Next the target stacking instructions defined above are generated. Finally, a C_KILL_EXCEPT instruction is generated. When the M-Machine executes the C_KILL_EXCEPT instruction, it will use all stacked values above the stack mark as its keep list and delete all

local variables not on this list. After deletion is completed, all target specifications and the stack mark are deleted.

Program Transfers

Three Mumps commands transfer control from the current location to a target location: DO, GOTO, and XECUTE. The DO command is the Mumps equivalent to a subroutine call. Upon completion of the target code, control is returned to the invoking location, which is either the next DO argument or the next command. The GOTO is an unconditional transfer. Control is never returned to the issuing location. The XECUTE command causes the execution of one line of Mumps code that is contained in either a local or global variable. Like the DO command, when the target code completes, control is transferred to the next XECUTE argument or the next command.

Also included in this section is the ZXtended added program mode command that is used to invoke the M-Machine. Although it is a C-Based Mumps command, it is presented here because it is a program transfer, behaves similarly to a DO command, and uses the same control structure as do the other program transfer commands.

Control stack. All program transfer commands within the C-Based interpreter are controlled by a stack, the gosub stack. The stack organization was chosen for its ability to preserve return paths and permit recursion. The M-Machine also employs a stack, the control stack, for the execution of transfer commands. Besides internal use by the M-Machine, the control stack is used by the C-Based Interpreter when the

M-Machine is invoked via the ZX command and in the performance of an XECUTE command on behalf of the M-Machine. The fields of the control stack follow.

- File Name — The base file name is that of the program loaded when the M-Machine is invoked. The base program will be replaced whenever an external GOTO is executed. For DO commands, the file name entry is that of the invoked program.
- Return Address — The address of the intermediate instruction where execution is resumed when a DO or XECUTE completes.
- Command Type — Flag indicating the current state of the M_MACHINE and/or transfer command being executed. Possible values and interpretations are: M_NONE, initial value, M_IDLE, program loaded but M-Machine inactive, ZX, M-Machine executing, M_DO, entry for each active DO command, and M_XECUTE, entry for XECUTE command.

ZXtended added program mode command. The M-Machine can be invoked from either direct or program mode by the ZXtended command. One argument is required: the file name of the program to execute. In behavior, the ZXecute is similar to a DO command. Quits, when no DOs or FORs are active, terminate the M-Machine but not the invoking C-Based Mumps program. Halts however, will result in termination of both the M-Machine and the C-Based interpreter. The target intermediate program may be either a complete program or subprogram.

The C-Based interpreter initializes the command field of the control stack to M_NONE when the interpreter is initialized and after every error reset. When the first ZXtended command is executed, the interpreter defines the base file name, loads the target file, and starts the M-Machine. When the M-Machine terminates normally, the base

program is preserved and the command field of the control stack is set to M_IDLE. Any subsequent invocation will compare the loaded program to the target program and bypass the program load if possible.

One implementation caveat concerns the specification of program file names. The accepted file extension for Mumps program files is "mps". The object files generated by the translator have the extension "obj". Program files that invoke other program files could have the extension "mps" hard coded within them. Thus the M-Machine would not be able to locate the translated program unless the hard coded extensions were changed to "obj". This would require duplicate sources, one for regular Mumps and one for intermediate Mumps. To alleviate this situation, whenever the M-Machine attempts to load a file with the "mps" extension, the extension is automatically changed to "obj" and the file is loaded.

DO command. The DO command requires one argument, the entry point of the subprogram to be invoked. Entry point specifications have a variety of forms. However, all forms are a combination of: program label, offset from the label, and filename. The argument may be either coded as a constant or may be contained in a variable (indirection). The offset forms are not implemented in either C-Based Mumps or in the M-Machine. The translator recognizes and discards any specified offsets. Without offsets, the forms become: internal label, external filename, or external label plus filename.

There are three specialized instructions used to execute a DO. These are C_DO, C_TRANSFER, and C_INDIRECT_TRANSFER. In addition, depending on the argument form, the various fetch instructions and C_BRANCH may be

used. Using these instructions, the translator, depending on the argument form, will generate one of two possible sequences. These are illustrated in Figures IV-3 and IV-4. The translator is capable of generating the abbreviated internal branch sequence for hard coded constants shown in Figure IV-3, because the target address is known to the translator and can be resolved during translation. Indirectly specified internal labels can not be resolved during translation because the target is not known to the translator.

In both sequences, the C_DO instruction is used to prepare the control stack for the transfer. The control stack contents after C_DO execution are illustrated in Figure IV-5. At the beginning of the C_DO instruction, the top entry of the control stack contains the current program file name. The command type field is defined but is not relevant to the C_DO. First the C_DO defines the return address field as the address of the second instruction following the C_DO. The next instruction following the C_DO is the actual transfer instruction: C_BRANCH, C_TRANSFER, or C_INDIRECT_TRANSFER. The formula for the return address computation is PC + two + length of next instruction. The two is for the length of the C_DO instruction. Next, the control stack is pushed and the file name propagated to the new top. The propagation is done in anticipation of performing the internal transfer sequence in Figure IV-3. The file name will be redefined if an external sequence as shown in Figure IV-4 is performed. Finally, the command type of the new top entry is marked with the DO command type identifier, M_DO.

The transfer instructions, C_TRANSFER and C_INDIRECT_TRANSFER have two functions. The first is to resolve the target entry point. The

Address	Instruction
program counter (PC)	C_DO.
PC + 2	C_BRANCH.
PC + 6	Return instruction.

Figure IV-3. DO Command Instruction Sequence for Constant Specified Internal Labels.

Address	Instruction
PC - 4	Appropriate fetch instruction for constant, local variable, or local array.
PC	C_DO.
PC + 2	The transfer instruction.
PC + 4	Return instruction.

Figure IV-4. DO Command Instruction Sequence for Indirectly Specified Internal Labels and all External Specifications.

	File Name	Return Address	Command Type
Top:	Invoked program	M_UNDEFINED	M_DO
Top-1:	Invoking program	C_DO + length of transfer instruction.	By prior definition.

Figure IV-5. Control Stack after Execution of C_DO Instruction.

second is to load the program file if it is not the same as the current program. The difference between the two transfer instructions is in the way they process the target specification. In the C_INDIRECT_TRANSFER instruction, the indirect target specification must be parsed (by calling the interpreter routine, parse_()) before it is processed. In the C_TRANSFER, the parse was completed during translation. Otherwise the two transfer instructions are identical and, in fact, share most of the actual implementation code. For the balance of this discussion, C_TRANSFER will be a reference to both transfer instructions.

The C_TRANSFER instruction pops the target entry point from the general purpose stack and separates it into label and file name parts. If the file name is defined, the file is loaded and the file name is stored in the file name field of the control stack. Next the transfer address is determined. If a label was specified, the label section is searched and the code address is retrieved. If no label was specified, the transfer address is code beginning. The computed transfer address is loaded into the program counter and execution is continued. An error is reported if the target entry point cannot be resolved.

When a DO terminates normally, the DO invocation is popped from the control stack, the invoking file is reloaded, if necessary, and execution resumes at the return point. Both normal and premature terminations are discussed in more detail when the QUIT and HALT commands are presented.

GOTO command. The exact behavior of the GOTO is dependent on the form of the argument being executed. The possible forms for the argument

are identical to the DO argument forms presented above. If the argument is an internal label, the GOTO cancels all active FORs, leaves all active DOs unaffected, and transfers control to the target location. In this case the control stack is not altered and the M_Machine simply performs an internal branch. When the argument is an external form, either filename or label plus filename, all FORs are cancelled, the control stack is cleared back to its base, thereby cancelling all DOs, and the specified file becomes the new base program. The target file is loaded into memory; the entry address is resolved; and the execution is resumed at the target location.

The intermediate instructions necessary to perform a GOTO are identical to those of the DO command except for the addition of the C_GOTO instruction. In terms of instruction sequences, the sequences presented in Figures IV-3 and IV-4 are used with the substitution of the C_GOTO instruction for the C_DO instruction. The C_GOTO's only action is to cancel all active FOR loops. The rest of the GOTO is handled by the transfer instructions. Besides the actions specified earlier, the C_TRANSFER (and C_INDIRECT_TRANSFER) prepares the control stack properly when the transfer is the result of a GOTO command. After the target specification is processed and it is found to be an external specification, the C_TRANSFER tests to see that a GOTO is in progress. In this case, the control stack is cleared back to its base and the target specification becomes the new base program. The transfer then proceeds with the file load, address resolution, and control transfer.

XECUTE command. The XECUTE command, executes one line of Mumps code. The code is contained in its argument variable or expression. In behavior it is similar to the DO command. When execution of the target code completes, the execution of the invoking code is resumed at either the next XECUTE argument or the command following the XECUTE. Unlike the DO command, the target code does not have an explicit entry point. Instead, the contents of the argument variable or argument expression are processed and treated as if they were the next line of code. C-Based Mumps permits the target code to contain any valid Mumps code. Because the target code is unknown at translation time and may be dynamically changed during the course of the programs execution, the target code of an XECUTE is interpreted by the C-Based interpreter rather than the M-Machine.

The M-Machine, depending on the type of variable, uses either a fetch instruction sequence or an expression evaluation sequence to process the target code. This is followed by the C_XECUTE instruction which passes control to the C-Based interpreter. Before transferring control to the C-Based interpreter, the C_XECUTE must prepare numerous control structures. First, the C_XECUTE modifies the C-Based interpreter's structures to look like an XECUTE was encountered by it. This includes pushing the gosub stack, preserving the contents of the current command line on the command stack, appending the target code to the end of program space and setting the program counter, pdlcur, to the first character of the target code. When the gosub stack is pushed, its command type field is defined with M_XECUTE flag. This flag will cause the proper return to the M-Machine when execution of the target code is

completed. After preparing the interpreter's structures, the C_XECUTE pushes the M-MACHINE control stack. The command type field is defined as M_XECUTE and the return address is defined as the instruction immediately following the C_XECUTE instruction. Finally the C_XECUTE returns to the C_Based interpreter.

When the C-Based interpreter gains control, it immediately begins executing the target code. When the target code completes, either by normal completion or prematurely by a QUIT command, the C-Based interpreter performs the actions of a normal XECUTE return. This includes restoring the command line, resetting pdlcur, and popping the gosub stack. However, instead of returning to the normal XECUTE return point, the M_XECUTE flag causes the interpreter to prepare the M-Machine for resumption of execution. This involves loading the PC with the return address and popping the control stack. The M-Machine is then called and execution resumes at the instruction immediately following the C_XECUTE instruction.

While the C-Based interpreter is performing an XECUTE on behalf of the M-Machine, an error will be reported if a ZXtended command is attempted. This situation is detected by checking the command type flag on top of the control stack when the ZXtended is attempted.

Conditional Execution

Mumps IF and ELSE commands provide for the conditional execution of the remainder of the command line. Both commands test the built in variable, $Test. The IF command executes the remainder of the command line if $Test is true. When $Test is false, the remainder of the current

command line is skipped and execution continues with the next command line. The ELSE is just opposite. The ELSE command executes the remainder of the command line when $Test is false. The IF command may also be used to set $Test.

IF commands may have none, one, or many logical expression arguments. In the no argument form, the current value of $Test is used to determine the action taken. In the one argument form, the expression is evaluated, $Test is set, and the IF test performed. In the multiple argument form, the evaluation sequence is repeated for each argument. In this case, the IF will fail if one of the arguments evaluates false. The setting of $Test will reflect the result of the last argument evaluated. ELSE commands do not have arguments. Like the IF with no arguments, they use the current value of $Test to determine the action taken.

There are three specialized instructions for executing IF and ELSE commands: C_IF, C_ELSE, and C_SET_$TEST. In addition, the general expression evaluation instructions are used to evaluate the logical expressions. Both C_IF and C_ELSE have as an operand the address of the first instruction of the next command line. When either is executed, the current setting of $Test is tested. If the test fails, the the branch to the operand address is taken. Otherwise, execution continues with the next instruction.

The instruction sequence for IFs without arguments and ELSEs is simply the C_IF or C_ELSE instruction, respectively. An IF with one argument uses an instruction sequence consisting of expression evaluation, C_SET_$TEST, and C_IF. An IF instruction with multiple arguments requires the repetition of the previous three part sequence,

once for each argument. The multiple argument case is illustrated in Figure IV-6. The operand of each C_IF in the multiple argument case is identical and is the address of the next command line.

Post conditionals. A post conditional is a construct very similar to an IF statement. It consists of a logical expression that is attached to either a command or command argument. When a post conditional is encountered, the expression is evaluated and, if the result is true, the command or argument is executed. If the result is false, the command or argument is skipped. Execution then continues with the next command or argument. Figure IV-7 illustrates the instruction sequence for a DO command that has both the command and argument post conditionalized. The commands and arguments for which post conditionals are allowed are given in [1].

The instruction sequences used to perform post conditionals are almost identical to those of the IF. Post conditionals do not set $Test and the failure branch address encoded in the C_POST_IF is different. The C_POST_IF instruction finds the logical expression result on top of the stack rather than testing $Test. For C_POST_IF operands, the translator encodes the address of the first instruction of the next command or command argument, whichever is applicable. The code to execute the post conditional precedes the instructions for the command or command argument. When the C_POST_IF fails and the branch is taken, the command or command argument is skipped.

	Instruction	Comment
Argument_1:	Expression Evaluation	
	C_SET_$TEST	
	C_IF	Operand is Next_Line: address.
Argument_2:	Expression Evaluation	
	C_SET_$TEST	
	C_IF	Operand is Next_Line: address.
Line_Remainder:	Code for remainder of command line following IF.	
Next_Line:	Code for next command line.	

Figure IV-6. Instruction Sequence for IF Command with two Arguments.

	Instruction	Comment
Command_Post:	Expression Evaluation	
	C_POST_IF	Operand is Next_Command address.
Argument_Post:	Expression Evaluation	
	C_POST_IF	Operand is Next_Command: address.
	C_DO	
	C_BRANCH	
Next_Command:	Code for command following DO.	

Figure IV-7. Instruction Sequence for Internal Form of DO Command where both the Command and Argument are Post Conditionalized.

FOR Command

The FOR is probably the most complex command to translate and execute. Unlike loop commands in most languages which allow for only one loop control specification, Mumps FOR loops permit multiple loop control argument specifications. For arguments have three forms: scalar, iterative infinite, and iterative with limit.

The extent of a FOR loop is the remainder of the command line. Additional commands can be included within the loop by invoking them via a DO command. FOR commands may be nested. Scalar values may be text or numeric. Iterative values should be numeric, but if they are text, a numeric interpretation will be forced on them. Increments may be positive or negative. The sign of the increment determines the direction of the limit test: greater than or equal to for negative increments and less than or equal to for positive increments.

A FOR executes as follows. If the argument is a scalar, it is assigned to the loop control variable (LCV) and the loop is executed. Iterative arguments are handled in the normal way. If a limit value is specified, the test is at the top of the loop and is performed before both the first and each subsequent loop traversal. It should be noted that the iterative form without a limit value is an infinite loop that can only be terminated by a HALT, QUIT, or GOTO command. The arguments are executed serially. When an argument completes, the LCV is reinitialized and the next argument executed, or if no more arguments are present, the program continues with the next command line.

The M-Machine controls FOR execution with a specialized stack structure, the for stack. When a FOR is entered, its control information

is pushed unto the for stack. When the FOR exits, the stack is popped. The top of the stack always represents the innermost FOR. A structure very similar to the one presented here is used in the C-Based interpreter to control FOR loops.

The for stack contains the following fields:

- Exit address. It is used when the loop is prematurely terminated by a QUIT command.
- Return address. The return location at the top of the loop where control processing resumes after executing the loop body. This value changes between arguments.
- Address of the beginning of the loop body.
- Address of loop control variable in variable name section.
- Current value of loop control variable. Used primarily to pass the current value between incrementing and testing instructions.
- Increment value. The expression is evaluated before the first loop traversal of an argument.
- Limit value. The expression is evaluated before the first loop traversal of an argument.

The basic arrangement of the executable code that controls a FOR loop is illustrated in Figure IV-8. The M-Machine instruction that accomplishes each step is given beside the step. Figures IV-9, IV-10, and IV-11 illustrate the executable code for scalar, iterative infinite, and iterative with limit arguments respectively.

There are some minor points in the implementation of FOR commands worth mentioning. In the scalar code, the use of current loop control variable value field is unnecessary. Other instructions could have been used. It was done this way to make the scalar code consistent with the other argument forms. In the iterative forms,this field is necessary to pass values between incrementing and testing. It originally was

Instruction	Comment
F_INC_FSX	Increment for stack pointer.
F_DEF_EXIT	Define loop exit address.
F_DEF_LCV	Define name of the loop control variable.
	Code for 1st loop control argument.
	Code for 2nd loop control argument.
.	
.	
.	
	Code for last loop control argument.
	Code for loop body.
F_RETURN	Return to loop top using the return address.
F_DEC_FSX	Exit loop by decrementing for stack pointer.

Figure IV-8. Instruction Sequence for the General Form of a FOR Loop.

Instruction	Comment
	Evaluate scalar argument expression.
F_DEF_CURRENT	Define the initial value of the loop control variable in both the for stack and the local symbol table.
F_DEF_RETURN	Define return address as either (a) or (b): a) Address of next loop control argument if present. b) FOR exit if last argument.
F_LOOPBODY	Branch to loop body. This step is omitted if this is last FOR argument because the loop body will immediately follow.

Figure IV-9. Instruction Sequence for a FOR Scalar Argument.

Instruction	Comment
Setup:	
---	Evaluate initial expression.
F_DEF_CURRENT	Define current value of loop control variable in both the stack and local symbol table.
---	Evaluate increment expression.
F_DEF_INCREMENT	Define increment value in control stack.
F_DEF_RETURN	Define return address as that of Increment:
F_LOOPBODY	Branch to loop body.
Increment:	
F_INCREMENT	Fetch current value of loop control variable, increment using increment value in stack, and store result in both the stack and symbol table.
---	Fall through into loop body. No other arguments follow this form because they cannot be executed. The compiler recognizes this case and does not generate code for additional arguments.

Figure IV-10. Instruction Sequence for an Iterative Infinite Argument.

Instruction	Comment
Setup:	
---	Evaluate initial expression.
F_DEF_CURRENT	Define current value of loop control variable in both the stack and local symbol table.
---	Evaluate increment expression.
F_DEF_INCREMENT	Define increment value in control stack.
---	Evaluate limit expression.
F_DEF_LIMIT	Define limit value in control stack.
F_DEF_RETURN	Define return address as that of Increment:
C_BRANCH	Branch to Limit_Test:
Increment:	
F_INCREMENT	Fetch current value of loop control variable, increment using increment value in stack, and store result in both the stack and symbol table.
Limit_Test:	
F_LIMIT_TEST	Perform limit test and branch to (a) or (b) a) Test passed: loop body b) Test failed: next argument or loop exit address if last argument.

Figure IV-11. Instruction Sequence for an Iterative with Limit Argument.

intended as a cache to eliminate the fetch before incrementing. As it is here illustrated and as it is in the C-Based interpreter, the loop control variable can be changed during loop traversal and will effect loop execution appropriately.

Another point concerns the existence of the loop body address in the control stack and the instructions: F_DEF_LOOPBODY and F_LOOPBODY. This field could have been eliminated by using C_BRANCH instructions to replace F_LOOPBODY instructions and by having a two operand F_LIMIT_TEST instruction. The presented implementation was chosen to minimize the number of instruction types.

Termination Commands

C-Based Mumps has three termination commands QUIT, HALT, and ZExit. The action of the QUIT command is determined by context, while HALT and ZExit each have only one explicit definition. Somewhat related is the HANG command, which simply suspends the program for a specified time period (in seconds).

The contexts of the QUIT command are FOR, DO, and XECUTE. A QUIT within a FOR loop simply cancels the innermost FOR command containing the QUIT. The intermediate instruction that accomplishes this action is F_QUIT. This situation is recognized by the translator, which generates an F_QUIT for each QUIT command contained in a FOR loop. The F_QUIT instruction loads the program counter with the address of the containing FOR's prologue code.

In the DO case, a QUIT command causes return to the invoking code or interpreter direct mode, if only one DO is active. (Normally, program

mode is entered via the DO command.) In the case of an intermediate language program, entry is via the ZX command. For program consistency, the termination of the ZX command is handled identically to the termination of the DO command used to enter program mode. These actions are handled by the C_QUIT instruction. First, the command stack is inspected. If a DO is in progress (command type equals M_DO), C_QUIT performs a DO return by popping the control stack, loading the program counter with the return address field of the new control stack top, and reloading the program file if necessary. If no DO is in progress, the program in the M-Machine is terminating, and C_QUIT returns to the C_Based interpreter with a normal return code. At the end of each program file, the translator generates a C_QUIT instruction, which is the normal termination method of an intermediate code program. Upon return to the C-Based interpreter via a C_QUIT instruction, the interpreted Mumps program (if there is one) continues execution.

In the XECUTE case, a QUIT command causes return to the original command line. Since XECUTE commands are performed by the C-Based interpreter, this QUIT form is handled by the C-Based interpreter. Upon quitting an XECUTE, the C-Based interpreter checks to see if the M-Machine is active. If it is, the intermediate program is resumed.

The HALT command terminates the current program and returns the C-Based interpreter to direct mode. The intermediate instruction for performing this is C_HALT. This instruction causes the M-Machine to return to the interpreter with the M_HALT return code. The prologue code of the ZX command then returns the interpreter to direct mode.

The ZExit added program mode command embodies the action of a HALT. However, instead of returning to direct mode, the C-Based interpreter is terminated. The intermediate instruction is C_ZEXIT. This instruction returns to the interpreter, the M_ZEXIT return code. The ZX prologue code then terminates the C-Based interpreter.

The HANG command simply suspends program execution for a specified number of seconds. The intermediate instruction is C_HANG. The C_HANG pops the number of seconds to wait from the stack and performs a Unix or VAX C sleep() system call. Execution resumes when the sleep() system call returns.

Input and Output Operations

Mumps provides the usual input/output operations. This includes both file and the user terminal operations. Unlike many other languages, all data is transmitted as ASCII characters and strings. There is no provision for unformatted binary data. (The data type does not exist in C-Based Mumps). File operations are performed on units one thru four. The units are associated to files with the OPEN command. Unit five is the terminal and is always open for both reads and writes. The terminal is also the initial default unit. The target unit may be changed with the USE command. For files, this is done prior to the operation. Mumps provides three formatting controls: new line, new page, tab to specified column.

Output. All Mumps output is performed with the WRITE command. The target unit is the last unit specified with a USE command, or if no USE

has been issued, the user's terminal. The argument(s) to a WRITE command is a sequence of format controls and expressions separated by commas.

In the intermediate language there are five specialized instructions for the WRITE command. Three are for format control: C_WRITE_CRLF (new line), C_WRITE_FF (new page), and C_WRITE_TAB. Two are for data transmission: C_WRITE and C_WRITE_CHAR. C_WRITE is for string transmission and C_WRITE_CHAR is for transmitting non-printable ASCII, such as the Bell character. For single character write, the data is specified in decimal. A WRITE command instruction sequence generated by the translator consists of intermixed expression evaluation sequences and write instructions. Each write instruction immediately follows its associated expression evaluation sequence. C_WRITE_CRLF and C_WRITE_FF do not have an operand and simply perform their specified operation. C_WRITE_TAB, C_WRITE_CHAR, and C_WRITE all require a data value. For C_WRITE_TAB this is the number of spaces to tab; for C_WRITE_CHAR, the decimal value of the ASCII character; for C_WRITE, the string. All three instructions obtain the data by popping the result of the associated expression evaluation from the general purpose stack. In implementation, the M-Machine uses an interpreter service routine, inout(), to perform the actual write. The intermediate instructions first pop the stack (if data is required), setup data and flags for the call to inout(), and finally call inout().

Input. All Mumps input is performed with the READ command. In operation and form it is analogous to the WRITE command. Besides the basic read operation, the READ command can write before reading, can do

single character reads, and can read with a time-out. The write before read provides a convenient prompting mechanism by enabling quoted literal strings and the format controls to be written to the user's terminal during the course of a read command. This form is not allowed for units one thru four. The single character read (direct read) is a mechanism for inputting single characters without the necessity of a carriage return being entered. It is satisfied when the user types any character. The read with time-out is a standard read operation that must be completed within a specified time period (in seconds). If the time-out read is completed within the specified time, $Test is set to true and the target variable contains the input data. If the time-out expires before data is entered, $Test is set to false and the target variable contains nothing.

The intermediate language uses three specialized read instructions: C_READ, C_READ_DIRECT, C_READ_TIMEOUT. In addition, write before reads are handled by the expression evaluation and write instructions. The actual data storage is handled by the standard variable storage instruction sequences. The C_READ instruction uses the C-Based interpreter service routine, getstr1(), to perform the actual read. The value returned by getstr1 is then pushed on the general purpose stack. The read is then followed by the appropriate variable storage instruction which stores the read value in the target variable. C_READ_DIRECT and C_READ_TIMEOUT are done in an analogous manner. However their implementation is dependent on operating system calls to perform their specified functionality. In addition, C_READ_TIMEOUT obtains the time-out period from the general purpose stack.

Support commands. There are three input/output support commands: OPEN, CLOSE, and USE. The OPEN command associates an integer unit number (one thru four) with a filename and opens the file for either input or output, but not both. The CLOSE command closes the file associated with a unit number and frees the unit for other uses. The USE command makes the specified unit the current target of all READ and WRITE commands. There is a built-in variable, $IO, that can be queried to determine the current active unit.

The intermediate language implements these three commands with the C_OPEN, C_CLOSE, and C_USE instructions. The C_OPEN command requires two parameter values: the unit number and the complete filename. These parameters are produced by expression evaluation sequences. The C_OPEN then obtains the values from the general purpose stack. The filename parameter requires additional processing. Whether the file is opened for read or write is dependent on a switch (/NEW or /OLD) that is appended to the filename. Output files must be opened /NEW which creates a new file. Input files must be opened /OLD which implies the file already exists. After the C_OPEN attempts the actual file open, it uses the result to set $Test to indicate the success or failure of the open. Both the C_CLOSE and C_USE instructions require as a parameter, the unit number. This value is placed on the stack by an expression evaluation sequence. C_CLOSE obtains the unit from the stack, closes the associated file, and marks the unit number as free in an internal data structure. C_USE obtains the unit from the stack and sets an internal control variable with the unit value. All input/output control variables are common to both the C-Based interpreter and the M-Machine.

Additional C-Based Mumps Program Commands

The C-Based Mumps interpreter includes, as language extensions, three added program commands. These are ZP, which is the logic interpreter interface, ZG, which initializes the global array database, and ZExit, which is the exit command discussed previously.

The ZP command has one argument, the logical predicate to be evaluated. In C-Based Mumps, logical predicates are global array specifications. Before invoking the logic interpreter, the C-Based interpreter processes the global specification into the coded form discussed previously. The specification is then passed to the logic interpreter for evaluation. Upon completion, the logic interpreter returns a value of "1" when the predicate evaluates true and "0" otherwise. The returned value is used to set $Test so that the running program may determine the evaluation result. The logic interpreter also may return, through the local symbol table, predicate variables that have been defined in the course of a successful rule evaluation. This, however, is transparent to the C-Based interpreter. The ZP command is performed in the intermediate language by the specialized C_ZP instruction and the general instructions that build array specifications. The translator first generates the instructions that build the array specification on the general purpose stack. This sequence is followed by the C_ZP instruction. C_ZP first builds the global specification from the stack and then calls the logic interpreter. When the logic interpreter completes, C_ZP then sets $Test using the logic interpreter return value. The logic interpreter also

returns error conditions which are immediately passed back to the C-Based interpreter and cause the intermediate program to be terminated.

The only operation performed by the ZG command is to initialize the global array database. This is a stand alone operation and is completely self contained. In the intermediate language, this is performed by the C_ZG instruction, which is simply a copy of the code found in the regular interpreter.

CHAPTER V

PROJECT RESULTS

As stated in Chapter I, the goal of the project was to investigate the potential for program execution speed improvement by the use of an intermediate language to eliminate parsing overhead. The design was constrained by the requirement that the intermediate language be functionally identical to the fully interpreted version. The project required the following three activities.

- The design of the intermediate language.
- The implementation of a translator to generate the intermediate language. This was accomplished by converting a copy of the C-Based interpreter to be a stand-alone translator for the intermediate language.
- The implementation of an intermediate language interpreter and its incorporation into the C-Based interpreter.

Overall Results

All stated goals of the project have been met. A fully compatible, transparent intermediate language was designed and implemented. The test results presented in the following sections show that the elimination of parsing overhead by the use of an intermediate language does result in execution speed improvements that are directly related to the amount of parsing required to execute a given Mumps command or language feature.

The C-Based Mumps software used for this project was a UNIX test version authored by Dr. Kevin O'Kane. This interpreter was ported to a Digital Equipment Corporation MicroVAX II running MicroVMS Version 4.2. All software was compiled using Digital's VAX C Version 2.1. The port from UNIX to VMS disabled a few language features that make heavy use of

UNIX system calls. The features are the JOB command, direct READ, and READ with time-out. These features have been considered in the design of the intermediate language but are not implemented in the VMS version.

In the implementation of the intermediate language interpreter, code efficiency was not pursued beyond the level of good programming practice. No attempt was made to substitute faster in-line code for subroutine calls or to take architectural advantage of the MicroVAX II.

In all results presented in this chapter, the translation cost (time) in not included in the intermediate language execution times.

Performance Evaluation

The relative performance between the C-Based Mumps interpreter and the intermediate language interpreter will be measured at the command and program fragment level rather than at the program level. This approach was chosen for the following reasons.

- It would be extremely difficult to construct a fair and representative set of test programs.
- Measuring performance at the program level would not provide results that could be used in evaluating the instructions or instruction sequences that implement specific Mumps commands.
- Performance of both the C-Based and intermediate interpreters are heavily influenced by string lengths and symbol table contents. Both of these influences are outside the area addressed by this project.

The effect of string length and symbol table fullness on program performance is substantial. Both the internal and external data type of C-Based Mumps is string. This includes variable names, labels, and all operand strings. In later sections it will be seen that as string length increases, execution time also increases. The same relationship is true

regarding the symbol table density. As the number variables in the table increases so does execution time.

Because both the intermediate interpreter and the C-Based interpreter share the same symbol table, data manipulation, and support routines, the external effects must be controlled in order to achieve a fair appraisal of the intermediate interpreter. For example, in an environment with an empty symbol table and using minimal string lengths, an intermediate program fragment may execute in one second while the fully interpreted code takes one and a half seconds. As symbol table density and string length are increased, execution times may increase to three seconds for the intermediate and three and a half seconds for the fully interpreted. In both cases, the performance improvement from the intermediate language is one half second but the percentage improvement falls from 33% to 14%.

In all tests the results are CPU execution times and are expressed in seconds and hundredths of a second. The performance percentage increase was obtained by the following formula:

$$\frac{\text{Mumps execution time - intermediate execution time}}{\text{Mumps execution time}} \times 100$$

Base program fragment tests. In order to build up execution time, most tests require repetition. This is accomplished by a simple iterative FOR loop followed by the subject command(s). When the test involves a longer program fragment, the form becomes an iterative FOR loop that invokes a subroutine (DO command) containing the program fragment.

Three versions of the two base tests were performed. The results are shown in Table V-1. In the first version the local symbol table contains only the loop control variable. In the second version, ten, three character variables, each with a three character value are added to the symbol table. In the third version, the local symbol table contents are increased to one hundred variables.

In the simple FOR loop test the intermediate form is shown to be only marginally faster. This is due to the fact that the C-Based interpreter only parses the argument expression once. The values are then kept in a table. The balance of the required manipulations are almost identical between the two.

In the subroutine test, the marked improvement is due to both the elimination of the DO parsing and label searching. The ability of the intermediate language to encode the location of internal labels will be shown to be even more dramatic when the number of labels is increased in a later test.

As discussed in Chapter IV, the implementation of the M-Machine saves the last program code that was loaded. In the above tests, execution improves five one hundreths to one tenth of a second (depends on program length) when the file load can be bypassed.

Expression evaluation and assignment tests. This section presents assignment tests of all three variable types, arithmetic expression evaluation tests, and built-in function call tests. Included in the function call tests is the invocation of the logic interpreter.

	Mumps (sec)	Intermediate (sec)	Improvement (sec)	(%)
Test 1. Simple FOR loop.				
FOR I=1:1:1000				
Symbol table empty	2.83	2.62	0.21	7.4
Ten variables	3.58	3.39	0.19	5.3
One hundred variables	5.87	5.68	0.19	3.2
Test 2. Simple FOR with subroutine call.				
FOR I=1:1:1000 DO A				
A QUIT				
Symbol table empty	3.83	2.94	0.89	23.2
Ten variables	4.58	3.70	0.88	19.2
One hundred variables	6.88	5.90	0.98	14.2

Figure V-1. Base Tests.

The results of the first group of tests are presented in Table V-2. These tests all involve the assignment of a simple constant expression to the various forms of Mumps variables. Test 1 is the assignment of one hundred three character constant values to one hundred three character local variables. In the C-Based based interpreter this requires two parses, one each for the left and right hand sides of the assignment statement. Although the parses were trivial, a 31.5% improvement was realized simply by their elimination. Tests 2 thru 6 study local array assignment with increasing array subscript complexity. The results range from 21.5% to 29.8% improvements. Because these results are comparable to simple scalar assignment in Test 1, it indicates that the increased parse complexity of the subscript evaluations is offset by the increasing subscript evaluation instruction sequence complexity in the intermediate language. The global array assignment tests, Tests 7 thru 9, are equivalent to the local array Tests 2 thru 4, respectively. For the global arrays the performance decreased to between 3.8% and 6.3% improvement. The percentage decrease is attributable to the order of magnitude decrease in parsing overhead resulting from reducing the number of loop transits from one thousand to one hundred. Because the global arrays are disk based storage, the execution time was dominated by the access method rather than interpretation.

Tests 1 thru 4 of Table V-3 present the results of arithmetic expression evaluation. The basic expression chosen for these tests is the one illustrated in Figure IV-1 on page 29. In Test 1 all values are two and three character constants chosen so that all intermediate results will be small whole numbers. Test 2 uses the same values but

	Mumps (sec)	Intermediate (sec)	Improvement (sec)	(%)
Test 1.				
Store 100 varaibles in the local symbol table.	0.73	0.53	0.20	27.0
Test 2.				
FOR I=1:1:1000 SET A("B")=1	4.66	3.63	1.03	22.1
Test 3.				
SET C="B" FOR I=1:1:1000 SET A(C)=1	5.12	4.02	1.10	21.5
Test 4.				
SET C(1)="B" FOR I=1:1:1000 SET A(C(1))=1	6.32	4.91	1.41	22.3
Test 5.				
SET A="ABC",B="DEF" FOR I=1:1:1000 SET LOC(A,B)="STR"	7.42	5.21	2.21	29.8
Test 6.				
SET A="ABC",B="DEF",C="GHI" FOR I=1:1:1000 SET LOC(A,B,C)="STR"	8.71	6.17	2.54	29.2
Test 7.				
FOR I=1:1:100 SET ^A("B")=1	3.36	3.20	0.16	4.8
Test 8.				
SET C(1)="B" FOR I=1:1:100 SET ^A(C)=1	3.38	3.25	0.13	3.8
Test 9.				
SET ^C(1)="B" FOR I=1:1:100 SET ^A(^C(1))=1	4.32	4.03	0.29	6.7

Figure V-2. Assignment Tests.

	Mumps (sec)	Intermediate (sec)	Improvement (sec)	(%)
Test 1.				
`FOR I=1:1:1000` `  SET K=(10+(14-12))*(144/12)`	34.42	31.83	2.59	7.5
Test 2.				

`SET A=144,B=12,C=14`
`FOR I=1:1:1000`
`  SET K=(10+(C-12))*(A/B)`
	35.40	33.37	2.03	5.7

Test 3.				
`FOR I=1:1:1000` `  SET K=(1+(4-2))*(9/3)`	23.15	20.77	2.38	10.3
Test 4.				
`FOR I=1:1:1000` `  SET K=(123+(413-229))*(911/387)`	129.64	127.28	2.36	1.8
Test 5.				
`FOR I=1:1:1000` `  SET J=$EXTRACT("ABCDEF",3,5)`	8.96	6.62	2.34	26.1
Test 6.				
`SET A(1)=1,A(2)=2,A(9)=9` `FOR I=1:1:1000 SET J=$NEXT(A(1))`	7.18	5.10	2.08	29.0
Test 7.				
`SET ^A(1)=1,^A(2)=2,^A(9)=9` `FOR I=1:1:1000 SET J=$NEXT(^A(1))`	20.30	18.05	2.25	11.1
Test 8.				

`SET K=33`
`FOR I=1:1:1000`
`  SET J=$SELECT(K<0:-1,K=0:0,K>0:1)`
	28.26	26.50	1.76	6.2

Test 9.				

`SET BP=-42.1,MP=-188,DEN=0.501`
`FOR I=1:1:25`
`  ZP ^CHEM("CHEM,BP,MP,DEN)`
	5.22	4.84	0.38	7.3

Figure V-3. Expression and Function Evaluation Tests.

they are stored in local variables rather than as constants. With an empty symbol table, the difference between Tests 1 and 2 is a decrease from an improvment of 7.5% to 5.7%. Test 3 is similar to Test 1 except that all values are only one character and all intermediate results are one character whole numbers. When Test 3 is compared to Test 1, execution time dropped for the C-Based interpreter dropped by 32.7% and by 34.7% for the intermediate language. The intermediate language improvement for Test 3 is 10.3%. The comparison between Tests 1 and 3 establish that execution time is dominated by string handling rather than parsing. This conclusion is further supported by Test 4, which uses 3 character constants that produce non-whole intermediate results. Comparing Test 4 to Test 3 we see execution time increasing by over 500% and the intermediate language improvement falling to 1.8%.

Tests 5 thru 9 in Table V-3 illustrate a variety of built-in function calls. Functions require a substantial amount of parsing to evaluate their argument lists and we expected to see substantial improvements. This is counter balanced, however, by the amount of processing required to implement the function. Test 5 uses $EXTRACT to extract a sub-string from a string. This operation does not require much processing and we see an improvement of 26.5%. Tests 6 and 7 both use the $NEXT function to find the next array subscript of either a local or global array. Here again we see a performance improvement decrease resulting from the disk accesses required by the global arrays. The improvement falls from 30.4% for the local array in Test 6 to 11.1% for the global array in Test 7. Test 8 illustrates a function, $SELECT that requires substantial internal processing. As expected we see an

improvement of only 5.0%. Test 9 is a test of the logic interpreter interface. The test is derived from an example given on page 44 of *C-Based Mumps User's Guide* [1]. The database is a simple chemical database and is as illustrated in the reference. The 7.3% improvement is consistent with expectations. To evaluate the logic rule, the logic interpreter accesses the disk database heavily. For this test it required over one thousand read operations which dominated execution time.

Control transfer and conditional execution tests. The results of this final group are illustrated in Table V-4. Test 1 tests both unconditional branching and conditional execution. The results show the best intermediate performance improvement, 54.9%. This is largely due to the ability of the intermediate language to directly encode the branch addresses for both the unconditional and conditional branches. Test 2 is comparable to the second base test in Table V-1 on page 5. In this test however, the subroutine target has two intervening labels. In addition string lengths are longer. Because of these factors, execution time for the interpreter increases but stays the same for the intermediate language. This results in the performance improvement rising for 23.2% in the base test to 31.9% in this test.

Tests 3 thru 5 examine the indirect performance of the XECUTE command. In addition, Tests 4 and 5 test the condition execution of Mumps post conditional expressions. To do the XECUTE, the intermediate interpreter must invoke the regular interpreter to evaluate the XECUTE argument. For this reason, the indirect code has been kept as

	Mumps (sec)	Intermediate (sec)	Improvement (sec)	(%)
Test 1.				

`SET I=1`
`Z IF I>100 QUIT`
`ELSE S I=1+1`
`GO A`
`B GO X`
`C GO Y`
`D GO Z`
`A GO B`
`X GO C`
`Y GO D`
	1.02	0.46	0.56	54.9

Test 2.				

`F I=1:1:1000 DO LABEL3`
`LABEL1 W !,"XXXXXXXXXX" QUIT`
`LABEL2 W !,"XXXXXXXXXX" QUIT`
`LABEL3 QUIT`
	4.32	2.94	1.38	31.9

Test 3.				
`FOR I=1:1:1000 XECUTE "QUIT"`	4.12	3.57	0.55	13.3
Test 4.				
`SET J=12` `FOR I=1:1:1000 XECUTE:J>10 "QUIT"`	5.92	5.36	0.56	9.5
Test 5.				

`SET J=12`
`FOR I=1:1:1000 XECUTE:J>10 "QUIT"`
`XECUTE:J>10 "QUIT":I<500 "QUIT":I>499`
	10.36	9.08	1.28	12.4

Test 6.				
`F I=1:1:100 W !,?10,"I=",I`	4.13	4.09	0.04	1.0

Figure V-4. Control Transfer and Conditional Execution Tests

simple as possible. The improvement which ranges from 9.5% to 13.3% is largely attributable to lack of parsing required in setting up the XECUTE target and in post conditional expression evaluation.

Test 6 writes a new line, tabs to column 10, writes a constant, and finally a variables value to the user's terminal. The insignificant performance gain is attributable to the fact that I/O requires significant overhead in the operating system which is outside the scope of the interpreter. In addition the same output routine is common to both interpreters.

Conclusion and Suggestions for Improvement

Although the elimination of parsing overhead does result in measurable performance improvement, the magnitude is not as large as desirable. Because of the empty symbol tables and short string lengths, the test results presented in this chapter are not realistic approximations of the normal program environment. Tests of some larger pieces of code indicate that the average improvement is on the order of 10%.

Now that parsing overhead is eliminated, the next step would be to take better advantage of the intermediate language virtual machine. One of the most dramatic performance increases was achieved by encoding branch targets in the internal transfer and conditional execution instruction sequences. This concept could be extended directly to the local variables. In fact it is already done but has not been taken advantage of. Both the fetch and store instructions directly address the variable name. The structure could be extended to include the variable's

value. It would also have to contain some control information, such as whether the variable is currently defined and whether its value has been updated in the symbol table. In essence, the intermediate interpreter storage would behave as a cache and would face the same coherency problems as the cache in a physical machine. Each time the intermediate interpreter was exited to perform an indirect evaluation, an XECUTE command, or a logic interpreter call, the main symbol table would have to be updated. Upon return to the intermediate interpreter, the internal storage would have to be restored to currency. The best approach to this problem would be to incorporate the intermediate symbol table into the symbol table manager. While the M-Machine is active, the symbol manager searches it for the symbol, then its own symbol table. When the M-Machine is deactivated, its symbol table is flushed.

The rendition of the intermediate language presented herein is at too high a level. That is, the decomposition has not included all of the problem. Part of this is due to the influence of the conversion of the C-Based interpreter to be a translator. It is felt by the author that for programming expediency, the conversion affected the intermediate language design. Too much attention was paid to the elimination of parsing overhead and not enough attention to other functional areas. It is quite possible that if the service routines, function calls, etc. were decomposed and included in the intermediate language, that substantial performance gains could be achieved. In the case of the logic interpreter, the global array subscript processing should be

included. Functions such as $SELECT, also could be included with the instructions already in the intermediate language.

In summary this project represents a very substantial effort and provided the author with an excellent learning experience in the area of language translation and intermediate machine and language architecture. To arrive at the point of being able to make performance tests, both the translator and intermediate interpreter had to be in a form of substantial completion. This made experimentation difficult. What this project best represents is a successful prototype for future experimentation towards dramatically improved interpreter execution speed.

LIST OF REFERENCES

LIST OF REFERENCES

1. O'Kane, K. C. *C-Based Mumps User's Guide*. University of Tennessee, Knoxville, Tn., 1985.

2. Ibsen, L. A Portable Virtual Machine for Ada. *Software--- Practice and Experience 14,* 17-29, (1984).

3. Kornerup, P., Kristensen, B. B., and Madsen, O. L. Interpretation and Code Generation Based on Intermediate Languages. *Software--- Practice and Experience 10,* 635-658, (1980).

4. Tanenbaum, A. S., Staveren, H., and Stevenson, J. W. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems 4,* 1(Jan. 1978), 21-36.

5. Kogge, P. M. An Architectural Trail to Threaded-Code Systems. *Computer 15,* 3(March 1982), 22-32.

6. Wulf, W. A. Compilers and Computer Architecture. *Computer 14,* 7(July 1981), 41-47.

7. Pleszkun, A. R. and Thazhuthaveetil, M. J. The Architecture of Lisp Machines. *Computer 20,* 3(March 1987), 35-44.

APPENDIX

APPENDIX

INTERMEDIATE LANGUAGE INSTRUCTIONS

This appendix lists all of the intermediate language instructions. For each instruction, the following information is given.

- The type of instruction as listed in Figure III-2, page 21.
- The interpretation of the operand field for single operand instructions.
- All actions performed by the instruction. This includes fields in control structures that are modified.
- All C-Based Mumps routines used during execution.
- Any errors that can occur during execution. The error numbers given are those defined by the C_Based interpreter.

A note on stacking. Many instructions result in a value being placed on a stack. However, if the value is stored in one of the sections of the object module, only the address is pushed onto the stack. If the value is dynamic and is not stored in the object module, the value string is stored in the heap section and the heap address is pushed onto the stack. In the instruction definitions, this explanation applies whenever stacking is performed.

Expression Evaluation and Symbol Table Management

These instructions are used in the course of the expression evaluation, assignment via the SET or READ commands, symbol table deletions with the KILL commands, and global locks. All operators are listed in a separate section entitled "Operators".

C_FETCH_CONSTANT

Type:	Single operand.
Operand:	Address of constant in the constant section.
Action:	Push the address of the constant onto the general purpose stack.
Routines:	None.
Errors:	None.

C_FETCH_LOCAL

Type:	Single operand.
Operand:	Address of variable name in local variable section.
Action:	Use the addressed variable name to look up value in local symbol table. The result is pushed onto the general purpose stack.
Routines:	sym_().
Errors:	17 - Variable not found.

C_STORE_LOCAL

Type:	Single operand.
Operand:	Address of variable name in local variable section.
Action:	Pop the value to be stored from the general purpose stack, and using the addressed variable name, store the value in the local symbol table.
Routines:	sym_().
Errors:	23 - Symbol table error. 31 - Program size.

C_FETCH_LOCAL_ARRAY

Type:	Single operand.
Operand:	Address of local array name in local variable section.
Action:	1) Build complete array specification as follows: a) Concatenate special code (206) for an open parenthesis to addressed local array name. b) Search stack until M_STACK_MARK located. c) Starting from the M_STACK_MARK, concatenate each subscript to the array specification. Subscripts are separated by the special code (207) for a comma. d) Concatenate special code (209) for a close parenthesis to array specification. e) Pop all subscripts from the stack. The top is now the M_STACK_MARK. 2) Using the completed array specification, look up its value in the local symbol table. 3) Replace M_STACK_MARK with the result of look up.
Routines:	sym_(), build_array().
Errors:	17 - Variable not found.

C_STORE_LOCAL_ARRAY

Type:	Single operand.
Operand:	Address of local array name in local variable section.
Action:	1) Pop value to be stored from stack. 2) Build complete array specification as described in C_FETCH_LOCAL_ARRAY instruction. 3) After array specification is built, pop the M_STACK_MARK from the stack. 4) Using the completed array specification, store the value obtained in step 1) in the local symbol table.
Routines:	sym_(), build_array().
Errors:	17 - Variable not found. 31 - Program size.

C_FETCH_GLOBAL

Type:	Single operand.
Operand:	Address of variable name in global variable section.
Action:	The steps are identical to C_FETCH_LOCAL_ARRAY except that the value is fetched from the global symbol table.
Routines:	global(), build_array().
Errors:	3 - Global not found.

C_STORE_GLOBAL

Type:	Single operand.
Operand:	Address of variable name in global variable section.
Action:	The steps are identical to C_STORE_LOCAL_ARRAY except that the value is stored in the global symbol table.
Routines:	global_(), build_array().
Errors:	None.

C_STACK_MARK

Type:	No operand.
Action:	Push the special integer constant, M_STACK_MARK, onto the general purpose stack.
Routines:	None.
Errors:	None.

C_INDIRECT

Type:	No operand.
Action:	1) Pop the indirect Mumps expression from the stack. 2) Call interpreter routine, parse_(), to evaluate the sub-expression. 3) Push result of parse_() evaluation on the stack.
Routines:	parse_().
Errors:	Error code returned from parse_().

C_KILL

Type:	No operand.
Action:	Delete the complete local symbol table by setting its length to zero.
Routines:	None.
Errors:	None.

C_KILL_LOCAL

Type:	Single operand.
Operand:	Address of variable name in local variable section.
Action:	Use the addressed variable name to delete the variable name and its value from the local symbol table.
Routines:	sym_().
Errors:	None.

C_KILL_LOCAL_ARRAY

Type:	Single operand.
Operand:	Address of variable name in local variable section.
Action:	1) Build the local array specification as described in the C_FETCH_LOCAL_ARRAY instruction. 2) Pop the M_STACK_MARK from the stack. 3) Using the array specification, delete that array element and all its decendents from the local symbol table.
Routines:	sym_(), build_array().
Errors:	None.

C_KILL_GLOBAL

Type:	Single operand.
Operand:	Address of variable name in global variable section.
Action:	The steps are identical to C_KILL_LOCAL_ARRAY except that the deletion is from the global symbol table.
Routines:	global(), build_array().
Errors:	None.

C_KILL_INDIRECT

Type:	No operand.
Action:	1) Pop the indirect target variable name expression from the stack. 2) Call interpreter routine, parse_(), to evaluate the variable name expression. 3) Using the variable name returned from parse, delete the variable (or array) from the local or global symbol table. The correct table is determined by the first character of the name ("^" indicates global).
Routines:	parse_(), sym_(), and global().
Errors:	Error code returned from parse_().

C_KILL_LOCAL_EX

Type:	Single operand.
Operand:	Address of variable name in local variable section.
Action:	Push the addressed variable name, which is a member of the keep list, onto the stack.
Routines:	None.
Errors:	None.

C_KILL_LOCAL_ARRAY_EX

Type:	Single operand.
Operand:	Address of variable name in local variable section.
Action:	1) Build the local array specification as described in the C_FETCH_LOCAL_ARRAY instruction. 2) Replace M_STACK_MARK on top of stack with the completed array specification, which is a member of the keep list.
Routines:	None.
Errors:	None.

C_KILL_INDIRECT_EX

Type:	No operand.
Action:	1) Pop the indirect variable name expression from the stack. 2) Call interpreter routine, parse_(), to evaluate the variable name expression. 3) Push the variable name returned from parse, which is a member of the keep list, onto the stack.
Routines:	None
Errors:	Error code returned from parse_().

C_KILL_EXCEPT

Type:	No operand.
Action:	1) Pop members of the keep list from the stack. The end of the keep list is determined by the occurence of M_STACK_MARK. 2) Pop M_STACK_MARK from the stack. 3) Delete all variables from the local symbol table that are not on the keep list.
Routine:	sym_().
Errors:	None.

Control Instructions

These instructions are used in the performance of all commands and features that control program execution. This include DO, GOTO, XECUTE, IF, ELSE, post conditional expressions, FOR, QUIT, HALT, and HANG.

C_BRANCH

Type:	Single Operand.
Operand:	Address of target instruction.
Action:	Load program counter with operand.
Routines:	None.
Errors:	None.

C_DO

Type:	No operand.
Action:	1) Define the return address in control stack (ctl_return) as current PC + 2 + length of next instruction. The instruction at PC + 2 is the calling instruction, which is either C_BRANCH, C_INDIRECT_TRANSFER, or C_TRANSFER. 2) Increment (push) the control stack pointer (ctlx). 3) Propagate the file field (ctl_file) from the stack top-1 entry to the stack top entry. 4) Define return field in stack top entry as M_UNDEFINED. 5) Define command field in stack top as M_DO.
Routines:	None.
Errors:	32 - Stack overflow.

C_GOTO

Type:	No operand.
Action:	Cancels all active FOR loops by setting the FOR control stack pointer (fsx) to -1.
Routines:	None.
Errors:	None.

C_TRANSFER

Type:	No operand.
Use:	Constant coded external DO, GOTO targets.
Action:	1) Pop target specification from stack. 2) Separate specification into label and file name components. In the process, any offset specified is discarded. 3) If GOTO being executed (determined by inspection of top entry on control stack) the control stack base is redefined. This cancels all active DO commands. 4) Define file name field in control stack top entry. 5) Load the target file. 6) Resolve label address if label specified. 7) Load program counter with transfer address.
Routines:	load_obj().
Errors:	8 - Label not found. 16 - Invalid expression. 26 - File error.

C_INDIRECT_TRANSFER

Type: No operand.

Use: Indirect DO and GOTO specifications. Target may be either internal or external.

Action:
1) Pop target specification from stack.
2) Separate specification into label and file name components. In the process, any offset specified is discarded. During separation, parse_() is called to evaluate the specification expression.
3) The balance is identical to C_TRANSFER except, that the transfer may be internal or external. For both internal DO and GOTO transfers, the label address is resolved and the program counter loaded with the transfer address. External transfers are as described under C_TRANSFER.

Routines: load_obj(), parse_().

Errors:
8 - Label not found.
16 - Invalid expression.
26 - File error.

C_XECUTE

Type: No operand.

Action:
1) Pop the indirect code from the top of stack and concatenate to the end of current loaded code in interpreter.
2) Prepare interpreter's "gosub" stack.
   a) Swap origin = 0.
   b) Save current program index.
   c) Save current command stack index.
   d) Save end of program pointer.
   e) Indicate current state is M_XECUTE.
3) Preserve interpreter's current program line in interpreter's command stack.
4) Set interpreter's program index to first character of target code.
5) Push control stack and define the return point as the next instruction (PC+2) and the command field as M_XECUTE.
6) Return to interpreter.

Routines: None.

Errors:
16 - Invalid expression.
32 - Stack overflow.

C_IF

Type:	Single Operand.
Operand:	Address of "test failure" target instruction. It is the first instruction of the next command line.
Action:	Test the $Test built in variable (tpx). a) tpx < 1 : Failure. Load PC with operand. b) tpx >= 1 : Success. Continue execution with next instruction.
Routines:	None.
Errors:	None.

C_ELSE

Type:	Single Operand.
Operand:	Address of "test failure" target instruction. It is the first instruction of the next command line.
Action:	Test the $Test built in variable (tpx). b) tpx > 0 : Failure. Load PC with operand. a) tpx = 0 : Success. Continue execution with next instruction.
Routines:	None.
Errors:	None.

C_POST_IF

Type:	Single Operand.
Operand:	Address of "test failure" target instruction. It is the first instruction of the next command or command argument.
Action:	Pop the stack and use the value obtained in the following test: a) Value = 0 : Failure. Load PC with operand. b) Value ≠ 0 : Success. Continue execution with next instruction.
Routines:	None.
Errors:	None.

C_SET_$TEST

Type:	No operand.
Action:	Pop the stack and use the value obtained in the following test: a) Value = 0 : Failure. $Test (tpx) = false. b) Value ≠ 0 : Success. $Test = true.
Routines:	None.
Errors:	None.

C_QUIT

Type: No operand.

Action: 1) If there are no active DO commands, the program is terminating.
   a) Set command field in control stack to M_IDLE.
   b) Set FOR stack pointer (fsx) to -1.
   c) Return to C_Based interpreter with normal termination return code.
2) If DO command is active (ctl_command = CTL_DO in control stack) then
   a) Load program file if ctl_file(ctlx-1) is different from current program file.
   b) Load PC with value in ctl_return.
   c) Pop control stack

Routines: load_obj().

Errors: 26 - File error.

C_HALT

Type: No operand.

Action: Return to the C-Based interpreter with the M_HALT completion code. The interpreter will then terminate all activity, restore its base program, and continue in direct mode.

Routines: None.

Errors: None.

C_HANG

Type: No operand.

Action: 1) Pop the number of seconds to hang from the stack.
2) Sleep for the specified number of seconds.
Non-numeric values result in zero seconds delay.

Routines: Unix sleep().

Errors: None.

C_JOB

Type: No operand.

Action: Not implemented because of VMS incompatibilities.

C_ZEXIT

Type: No operand.

Action: Return to the C-Based interpreter with the M_EXIT completion code. The interpreter will then return to the operating system without further activity.

C_ZP

Type:	Single operand.
Operand:	The address of a global variable name in the variable section. This global is the logical predicate.
Action:	1) Using the operand address, build the complete global array specification as described for the the C_FETCH_LOCAL_ARRAY instruction. 2) Using the global array specification as the logical predicate, call the logic interpreter, logic() 3) Set $Test based on the result of the logic interpreter's evaluation. True for evaluation success, false otherwise.
Routines:	logic().
Error:	As returned by logic().

C_ZGLOBAL

Type:	No operand.
Action:	Initializes the global array database. Any existing contents are destroyed.
Routines:	global(), Unix write();
Errors:	None.

FOR Instructions

The FOR instructions are specific to the FOR command. They are not used in the execution of any other commands. Many FOR instructions manipulate fields in the FOR control stack. The C declaration of this stack follows.

short fsx	FOR stack pointer.
short f_lcv[FOR_SZ]	Address of the loop control variable name in the variable section.
short f_lbody[FOR_SZ]	Starting address of the loop body.
short f_return[FOR_SZ]	Return address upon loop completion.
short f_exit[FOR_SZ]	Address of loop exit code.
char f_cur[FOR_SZ][20]	Current value of lcv.
char f_incr[FOR_SZ][20]	Value of increment expression.
char f_limit[FOR_SZ][20]	Value of limit expression.

F_INC_FSX

Type:	No operand.
Action:	Increments the internal FOR stack pointer, fsx. The base value for fsx is -1 when the for stack is empty.
Routines:	None.
Errors:	32 - Stack overflow (fsx >= FOR_SZ)

F_DEF_LCV

Type:	Single Operand.
Operand:	Address of loop control variable name string in variable section.
Action:	Move the operand to the f_lcv field.
Routines:	None.
Errors:	None.

F_DEF_CURRENT

Type:	No operand.
Action:	The result of evaluating the initial expression is on top of the general stack. This value is moved to the f_cur field and the general stack popped. Next, the lcv variable is assigned the initial value.
Routines:	SYM_().
Errors:	16 - Invalid expression. This occurs if no value is on general stack. 23 - Program size. 31 - Symbol table error.

F_DEF_INCREMENT

Type:	No operand.
Action:	The result of evaluating the increment expression is on top of the general stack. This value is moved to the f_incr field and the general stack popped.
Routines:	None.
Errors:	16 - Invalid expression. No value was on general stack.

F_DEF_LIMIT

Type:	No operand.
Action:	The result of evaluating the limit expression is on top of the general stack. This value is moved to the f_limit field and the general stack popped.
Routines:	None.
Errors:	16 - Invalid expression. No value was on general stack.

F_DEF_RETURN

Type:	Single Operand.
Operand:	Loop return address.
Action:	Move the operand to the f_return field.
Routines:	None.
Errors:	None.

F_DEF_EXIT

Type:	Single Operand.
Operand:	Address of loop exit code.
Action:	Move the operand to the f_exit field.
Routines:	None.
Errors:	None.

F_DEF_LOOPBODY

Type:	Single Operand.
Operand:	Address of first instruction of loop body.
Action:	Move the operand to the f_lbody field.
Routines:	None.
Errors:	None.

F_INCREMENT

Type:	No operand.
Action:	The lcv's current value is fetched, added to the the increment value (f_incr), and the result stored in the f_cur field. Next, the lcv variable is assigned the new value.
Routines:	SYM_(), ADD().
Errors:	17 - Variable not found. 23 - Program size. 31 - Symbol table error.

F_LIMIT_TEST

Type:	Single operand.
Operand:	Address of argument exit.
Action:	The lcv's current value (f_cur) is compared to the limit value (f_limit). If the comparison succeeds, the program counter is loaded with the contents of f_lbody. If the comparison fails, the program counter is loaded with the operand. The comparison is "greater than" for positive increments and "less than" for negative increments.
Routines:	numcomp().
Errors:	None.

F_LOOPBODY

Type:	No operand.
Action:	The program counter is loaded with the contents of the f_lbody field.
Routines:	None.
Errors:	None.

F_RETURN

Type:	No operand.
Action:	The program counter is loaded with the contents of the f_return field.
Routines:	None.
Errors:	None.

F_DEC_FSX

Type:	No operand.
Action:	Decrements the internal FOR stack pointer, fsx.
Routines:	None.
Errors:	None.

F_QUIT

Type:	No operand.
Action:	The program counter is loaded with the contents of the f_exit field.
Routines:	None.
Errors:	None.

Input and Output Instructions

The input and output instructions share a common data base with the C-Based interpreter. Most of the data base elements are arrays that contain the values applicable to each I/O unit. The I/O database variables are

short	io	The current unit number. It is the array index.
short	hor[]	The current horizontal position.
short	ver[]	The current vertical position.
char	opnflg[]	Current status and I/O direction.
FILE	*opnfile[]	Unix file pointer.

C_WRITE

Type:	No operand.
Action:	1) Pop the string to be output from the general purpose stack. 2) Set the control code for the interpreter routine, inout(), to indicate string output. 3) Call inout() to write the string.
Routines:	inout().
Errors:	26 - File error.

C_WRITE_CRLF

Type:	No operand.
Action:	1) Set the control code for the interpreter routine, inout(), to indicate that a carriage return/line feed pair is to be outputted. 2) Call inout() to perform the write.
Routines:	inout().
Errors:	26 - File error.

C_WRITE_FF

Type:	No operand.
Action:	1) Set the control code for the interpreter routine, inout(), to indicate that a formfeed is to be outputted. 2) Call inout() to perform the write.
Routines:	inout().
Errors:	26 - File error.

C_WRITE_TAB

Type:	No operand.
Action:	1) Pop the number of spaces to tab from the general purpose stack. 2) Set the control code for the interpreter routine, inout(), to indicate a tab is being output. 3) Call inout() to write the required number of spaces.
Routines:	inout().
Errors:	26 - File error.

C_WRITE_CHAR

Type:	No operand.
Action:	1) Pop the decimal value of the ASCII character from the general purpose stack. 2) Set the control code for the interpreter routine, inout(), to indicate ASCII single character output. 3) Call inout() to write the specified character.
Routines:	inout().
Errors:	26 - File error.

C_READ

Type:	No operand.
Action:	1) Call the interpreter routine, getstr1(), to read the value. 2) Push the value read onto the general purpose stack.
Routines:	getstr1().
Errors:	26 - File error.

C_READ_DIRECT

Type:	No operand.
Action:	Not implemented in this VMS version. The feature could be implemented with the VMS QIO system service.

C_READ_TIMEOUT

Type:	No operand.
Action:	Not implemented in this VMS version. The feature could be implemented with the VMS QIO system service.

C_USE

Type:	No operand.
Action:	1) Pop the unit number to uses as input output target from the general purpose stack. 2) Set internal variable, io, to unit value.
Routines:	None.
Errors:	14 - Argument list.

C_OPEN

Type:	No operand.
Action:	1) Pop the file name from general purpose stack. 2) Pop the unit number from stack. 3) Process file name to determine whether input (/OLD) or output (/NEW). 4) Perform open. 5) Set internal variables, opnfile[unit_number] and opnflg[unit_number] to reflect unit status. 6) Set $Test built-in variable (tpx) to reflect success of open operation.
Routines:	Unix fopen();
Errors:	14 - Argument list. 26 - File error.

C_CLOSE

Type:	No operand.
Action:	1) Pop the unit number to close from the stack. 2) Perform close. 3) Set internal variables, opnfile[unit_number] and opnflg[unit_number] to reflect unit free.
Routines:	Unix fclose();
Errors:	26 - File error.

Operator Instructions.

The Mumps operator instructions can be divided into two types. Unary, which require one operand and binary, which require two operands. One instruction of each type is illustrated. The other instructions are analogous in operation except for the support routine called and are simply listed.

Unary instructions:

C_UNARY_MINUS

Type:	No operand.
Action:	1) Pop the operand from the general purpose stack. 2) Multiply operand with minus one. 3) Push the result back on the stack.
Routines:	mult().
Errors:	16 - Invalid expression.

C_UNARY_NOT
C_UNARY_PLUS

Binary instructions:

C_DIVIDE

Type:	No operand.
Action:	1) Pop the divisor from the general purpose stack. 2) Check for divide by zero by comparing divisor to zero. 3) Pop the dividend from the general purpose stack. 4) Divide and push the result back on the stack.
Routines:	numcomp(), div().
Errors:	15 - Divide by zero. 16 - Invalid expression.

C_ADD	C_SUBTRACT	C_MULTIPLY
C_EQUAL	C_NOTEQUAL	C_GREATER
C_LESS	C_CONCATENATE	C_INTEGER_DIVIDE
C_MODULO	C_CONTAINS	C_FOLLOWS
C_PATTERN	C_NOTGREATER	C_NOTLESS
C_NOTCONTAINS	C_NOTFOLLOWS	C_NOTPATTERN
C_AND	C_NOTAND	C_OR
C_NOTOR		

Built-in and Z-Type Functions.

The majority of the functions are implemented by calling either the routine fcn() for built-in functions and variables and zfcn() for implementor added z-type. The exceptions are $DATA, $NEXT, $ORDER, $PIECE, and $SELECT. These are implemented directly by intermediate instructions and their operation is described below. The intermediate instructions for all other functions simply set an identification code, build an argument list (if there are arguments) by popping the stack, call the appropriate function module, and push the result back on the stack. These instructions are only listed for reference.

Standard functions:

C_$DATA

Type: No operand.

Action:
1) Build the argument list which is an array specification.
2) Depending on whether the argument is global or local, the appropriate symbol table manager is called to do the lookup processing.
3) Push the result on the stack.

Routines: sym(), global().

Errors: None.

C_$NEXT — Analogous to C_$DATA.

C_$ORDER — Analogous to C_$DATA.

C_$PIECE

Type: No operand.

Action:
1) Build the function arguments. Unlike array specifications, the arguments are build by finding the stack mark and popping back to the stack top.
2) Call fcn() to process.
3) Check for left hand side by testing an internal variable, setpiece. This variable is set by C_$PIECE_LHS. If it is set, left hand side processing is performed.
4) Push the result back on the stack.

Routines: None.

Error: 18 - Reference error.

C_$PIECE_LHS

Type: No operand.

Action:
1) Pop the left hand side value from the stack.
2) Indicate left hand side operations by setting the variable, setpiece to 1.

Routines: None.

Error: None.

C_$PIECE_LHS_SLA

Type: No operand.

Action: This instruction is identical to C_STORE_LOCAL_ARRAY except the value to be stored is under the array subscripts on the stack.

Routines: sym().

Errors: 23 - Symbol table error.
31 - Program size.

C_$PIECE_LHS_SGA

Type: No operand.

Action: This instruction is identical to C_$PIECE_LHS_SLA except the variable is stored in the global symbol table.

Routines: global().

Errors: None.

C_$ASCII	C_$CHAR	C_$EXTRACT
C_$FIND	C_$HOROLOG	C_$IO
C_$JOB	C_$JUSTIFY	C_$LEN
C_$RANDOM	C_$SELECT	C_$STORAGE
C_$TEST	C_$X	C_$Y

Z-type functions:

C_$ZALT	C_$ZCHANGE	C_$ZDATE
C_$ZERROR	C_$ZF	C_$ZG5
C_$ZINIT	C_$ZLIST	C_$ZMAKE
C_$ZNEXT	C_$ZPRINT	C_$ZSYSTEM
C_$ZTIME	C_$ZVARIABLE	C_$ZWAIT

VITA

Charles C. Ostrum was born in Harrisburg, Pennsylvania on April 22, 1949. He graduated from Lakeland Senior High in June, 1967. In June, 1976, he received a Bachelor of Arts degree with a major in Economics from The University of South Florida in Tampa, Florida.

From 1977 to 1980 he worked for the U. S. Department of Labor as an Economist and Investigator. From 1981 to 1983 he attended Graduate School at The University of Tennesse, Knoxville, majoring in Computer Science. In 1983 he began work at the U. S. Naval Research Laboratory, USRD, in Orlando, Florida as a Computer Systems Analyst. He resumed his graduate studies in 1986 and received the Master of Science degree in June, 1987.

The author is a member of the Association of Computing Machinery, and the Computer Society of the Institute of Electrical and Electronic Engineers.