This work was presented to the faculty of the Graduate School of Yale University in candidacy for the degree of Doctor of Philosophy. The author is presently in the Department of Computer Science at Yale University, New Haven, Connecticut 06520

A Machine Design for Efficient Implementation of APL

Charles Russell Minter

Research Report #81

May 1976

ABSTRACT

A MACHINE DESIGN FOR EFFICIENT IMPLEMENTATION OF APL

Charles Russell Minter

Yale University, 1976

This dissertation describes the organization of a computer system that efficiently executes APL programs. While the organization will perform well in a dedicated or single-user environment, it is best suited for use in time-sharing.

The computer system consists of two processors that share memory. One is a standard commercially available central processing unit, and the other is a unit specially designed to deal with dense, homogeneous, rectangular arrays. The computational subtasks of executing an APL program are divided between the two machines.

The CPU handles communication with the APL user; stores his program and data; and when called on to execute an APL program, allocates storage for any results or temporary results and divides the computations to be done into its own and the special processor's. It translates the relevant parts of the program into the form used by the special processor and cooperates with the special processor to assure the validity of each of the steps in the program.

The special processor performs those computations allocated to it by the central processor and cooperates with the central processor to determine the validity of the program. To minimize conflict in the use of the shared memory the special processor has its own small but very fast local memory, which contains the data it references often.

The work described in this thesis is a compromise between a software and a hardware approach to increasing the speed of execution of APL programs. The effectiveness of the design is examined using two simple programs. These studies suggest that the expected improvement over current APL implementations is substantial. The two-processor approach is also discussed in terms of its applicability to other problems.

ACKNOWLEDGEMENTS

I am indebted to many people for their help and encouragement while I was working on this dissertation. The staff of the Computer Science Department -- Polly Bobroff, Rosemary Browne, Cindy Benton, and Marie Clemens -- have given me a great deal of help in preparing the final copy of this thesis.

The members of my committee have all contributed to this work. I am most indebted to my advisor, Alan Perlis, who suggested the topic and, with his experienced hand, guided the work from beginning to end. Ned Irons provided insights that played an important role in the early stages of the work. Larry Snyder cited useful references along the way and his comments suggested new aspects of the problem.

The editorial comments of Mary-Claire van Leunen greatly improved an almost unintelligible first draft. Jim Meehan also made useful editorial suggestions. Martin Schultz offered specific comments on this work and some more general advise on writing a thesis.

The friendship, interest, and advise of George Schulz were invaluable to me, I will miss them dearly.

Finally I would like to thank the Mobil Research Foundation for providing financial support for this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The electronic stored-program digital computer was conceived and built to perform the immense number of calculations associated with certain scientific problems. An early computer could do in an hour what might take a mathematician months. At first, preparing a program was a tedious task; all programming was done in machine language. But, since so many computations were required to solve one problem and since the program remained the same for many problems (only the input parameters changed), the preparation time for the program was often small compared to the execution time [Goldstine 1972, Randell 1975]. Some early computer scientists realized that computers could be applied to a much wider class of problems if programs were easier to write.

Assembly programs, which allowed mnemonic substitutions for both instructions and data, eased the burden of writing a program. Libraries of programs were built to solve common problems [Mauchly 1947, Wilkes 1951a]. A programmer with a new problem could split it up and call on these library programs, called subroutines, to solve some parts of the problem. Assembly programs specified the solutions in the same detail used by the computer. Library subroutines eliminated some but not all of the detail. Even simple problems could require agonizingly intricate programs.

Probably the most important breakthrough in the history of the computer, aside from the organization of the machine itself, is the idea of a programming language [Sammet 1969, Rosen 1967]. No longer must a programmer specify machine instruction by machine instruction how to solve a problem; a programming language gives him a shorthand notation more suitable for the problem. Higher-level languages have radically increased the complexity and sophistication of problems computers can handle.

There are many programming languages, and the intended use of each most distinguishes one from another. Another characteristic is the logical separation between the programming language and the language of the target machine. In some languages the primitive operations and data structures correspond closely to the operations and data elements of a computer; these languages can be called machine-oriented. In other languages the operations and data structures seem natural to the user for some problems, but the machine must simulate them with a large amount of complex code. These can be called problem-oriented or user-oriented languages.

Although, as the name suggests, a user-oriented language makes it easier for a programmer to write a program, there are reasons for using a machine-oriented language. A machine-oriented language is so close to machine language that compilation is easier and generally

produces more efficient code. The work described in this thesis is an attempt to gain some of these benefits for a user-oriented language by designing a system to bring the language and the computer closer together.

Computer hardware has progressed since the first computer. The refinement of system organization and the development of components make modern computers as much of an improvement over the first generation as the first generation was over hand calculation [Goldstine 1972]. Solid-state technology will continue to have a tremendous impact on the evolution of computer systems by making their design easier and faster. Microprogramming has made implementing computer hardware similar to implementing software.

The primitive hardware operations in all digital computations are simple -- addition, negation, Boolean operations, data movement, and data selection. Even the simplest computer instruction requires several of these. Therefore it is natural to think of a computer instruction as a small program. In 1951 M. V. Wilkes argued that it was easier to design computers using microprogramming than to implement the higher-level instruction set directly [1951b, 1953].

It took fifteen years for Wilkes's ideas to become popular. Then a rapid shift to microprogramming resulted from the ready availability of solid-state components,

primarily read-only memory. A microprogrammed implementation of an instruction set requires more memory circuits and fewer combinatorial circuits than a direct implementation. At the time Wilkes put forth his design suggestions, the technology favored building combinatorial logic circuits rather than memory circuits; as a consequence, wide acceptance of microprogramming did not come until integrated circuit technology had shifted the favor from combinatorial to memory circuits [Signetics 1970, Signetics 1971]. The continuing advance of integrated circuit technology has simplified implementing a moderately complex machine so much that special purpose machines like the ones described here are feasible.

1.1 LANGUAGE-ORIENTED MACHINES

If the costs of the hardware and software in a computer system are used as the measure of progress in these areas, software has not advanced nearly as rapidly as hardware. Software costs for a computer system account for more of the total development cost than hardware, and the ratio is likely to increase. Programming languages help to decrease software costs by making it easier to write programs. Generally the higher the level of the programming language the easier a particular program is to write but the slower the program runs. One of the best uses of the improving hardware is to improve the efficiency of programs written in a high-level language. One way to do this is to design

special computer systems for high-level programming languages.

There have been a number of language-oriented machine designs, the first so early that the machine was implemented with relays [Burks 1954]. Computers have been designed with a wide variety of languages in mind. Even FORTRAN, whose primitive operations and data types are similar to those of a general-purpose digital computer, was the target language in some attempts [Melbourne 1965, Bashkow 1967]. The similarities are so great that it is unreasonable to expect that FORTRAN code will run faster on a FORTRAN machine than a machine-language translation of that code will on a regular computer. Indeed, Bashkow, Sasson, and Kronfeld [1967] cite as the rationale for the design of a FORTRAN machine the elimination or drastic reduction of compilation time. In essence they designed a hardware FORTRAN interpreter. Although the FORTRAN interpreter was not built, several hardware interpreters for BASIC, a language with primitives like FORTRAN's, have been built for use as programmable desk-top calculators.

For languages whose primitive elements are not closely related to the operands and operators of a digital computer, there is a trade-off between reducing the compilation time of a program and improving its execution time. That is, the more the machine language of the language-oriented machine resembles the higher-level language, the easier the

compilation. But some computations, like symbol table manipulations, are postponed from compilation time to execution time. The more complicated the machine language, the less efficient the execution.

Microprogramming has accentuated this trade-off by enabling a designer to implement complex software routines with a microprogram. It is possible to implement a standard interpreter in any microprogrammable machine. Substantial gains result; a program executed by a microprogrammed interpreter can take as much as a factor of ten less time than the same program executed by an interpreter written in the machine's macroinstruction set [Weber 1967]. For any microprogrammable computer and any interpretation algorithm, a microprogrammed interpreter is probably the best way to execute programs in a higher-level language. Preprocessing a higher-level language program can eliminate or reduce the work of interpretation.

1.2 A PROGRAMMING LANGUAGE

The programming language APL grew out of work done by Iverson [1962]. Polivka and Pakin [1975] and Gilman and Rose [1974] give descriptions of APL.SV, a popular APL interpreter. APL's notation was designed to resemble mathematical notation. The fundamental data types are scalars, vectors, and dense, rectangular, homogeneous arrays of numbers or characters. Logical data, integers, and floating-point numbers fall under the numeric data type.

APL variables are dynamic; a programmer need not know in advance how large the arrays in his program are going to be and he can change the size of an array many times during the execution of a program. There are a large number of operators in APL; some of them are familiar ones found in many programming languages while others are unique to APL. They all have been extended for use with array-structured data. Many of the primitive operations on arrays, not only arithmetic operations but also data rearrangement and data selection, are single operators. Therefore, APL performs many simple operations on arrays with a single operator, while some other programming languages need explicit control structure to do these operations. The APL program fragments that perform these operations are much shorter and more easily understood than the equivalent program fragments in other languages. The combination of the dynamic handling of data and the many powerful array-oriented operators make APL an attractive choice for the development of programs to solve problems that involve array-structured data.

Unfortunately there are some disadvantages to the use of APL [Abrams 1975]. If the data associated with a problem cannot be organized naturally into a dense, rectangular, homogeneous array, then data access and manipulation could well be inconvenient. For instance, lists and trees can be implemented in APL data structures but other languages provide them as primitive data types, and so are usually better suited for problems whose solutions are easily

expressed with lists and trees. Some languages require that many programs use explicit control structure. The power of the operators in APL enables some of these programs to be written without explicit control structure. For those programs that do require control, APL may not be the best choice. This weakness can be circumvented by writing functions that simulate the action of the missing primitives, but, just as before, at the cost of convenience and efficiency. As a consequence of these two disadvantages there are algorithms that are difficult to express in APL. These disadvantages are a result of the basic definition of the language; any attempt to alleviate them would require revising or adding to that definition. While such revisions might be beneficial, APL as it now stands is extremely powerful and is a good language for a wide class of problems.

This work focuses on another disadvantage. All production APL systems are interpretive, and as a result programs run more slowly than compiled code. Because of the dynamic character of the data, APL is not compilable in the same sense as FORTRAN. If the rank, size, or type of a variable is not known in advance, then the code generated for an operation involving that variable must be able to handle many very different situations. A compromise between compilation and interpretation is possible and that, along with some special hardware to ease both parts of the task, is what is being proposed here.

The original implementations of APL were all software interpreters. Perhaps one reason for the rapid acceptance of the language was the excellent quality of the implementation on the IBM 360 [Breed 1967]. APL has been the target of several language-oriented machine designs [Abrams 1970, Thurber 1970, Hassitt 1973, Grant 1974, Hassitt 1975].

Zaks et al. [1971] and Hassitt et al. [1973, 1975] developed APL machines by writing microprograms for existing microprogrammable machines; these machines are full or nearly full implementations of APL. Thurber and Myrna [1970] and Abrams [1970] proposed new hardware designs to support an APL machine; neither of these machines has been implemented. Abrams's proposed machine can be designed and built using technology available today, but it requires two machines. As Thurber and Myrna point out, their proposed design will probably be practical only when there are integrated circuits with many more components per chip. Both IBM and Micro Computer Machines make desk-top computers that use APL. In each of these machines APL is implemented by a hardwired microprogram [ED 1975, CD 1975].

Much of the work described in this thesis grew out of work begun by Abrams [1970] and some of the similarities are discussed in later chapters. In his dissertation Abrams gave a formal treatment of APL that showed that some operators, called the selection operators, could be applied

by changing a few parameters the data-access routine uses rather than by manipulating the data itself. Furthermore these operators exhibit certain relations similar to associativity in mathematics. Abrams's machine took advantage of the features he had demonstrated using techniques he called beating and drag-along to eliminate extraneous computations. Abrams designed his machine so that the machine language is close to APL; the result is that his machine is considerably more difficult to build than a machine with a more ordinary architecture. For instance, implementing the associative memory in his machine requires many integrated circuits. Abrams never intended to build a physical machine. This thesis describes work in which a number of compromises were made in the overall system and in the APL machine to make an implementation not only theoretically possible but also entirely practical.

1.3 OVERVIEW

Chapter 2 discusses general methods for sequencing through the elements in an array and describes how the ladder mechanism embeds code within a sequencing mechanism. Chapter 3 describes three techniques for using ladders to implement an APL system. One of these methods uses a specially designed computer and the chapter goes on to discuss three overall designs of this computer. Chapter 4 examines the design of two possible special machines in more detail and Chapter 5 estimates the cost and performance of

each. Chapter 6 discusses possible future work and the effect hardware developments will have on the general approach described in this thesis.

CHAPTER 2

ARRAY ACCESSING

Why might one expect substantial performance gains from constructing a special-purpose machine to execute APL? It is helpful to consider a simple example -- A←B+C.

In a scalar-based, compiled language this statement represents a very small amount of computation. If the variables are integers, the operations are among the most basic a general-purpose digital computer can do. The execution of this statement in APL is considerably more complex. An APL interpreter would probably begin by checking to see whether the statement is legal, whether the types, ranks, and shapes match appropriately. Next the interpreter would allocate storage for the variable A. Then it could begin the execution of the computation the statement specifies.

First the interpreter must fetch a data element from from B or C, say C; then it must fetch the corresponding data element from B; and then it stores the sum of these two data elements in the appropriate place in A. At this point the interpreter must check to see whether the computation the statement calls for has been completed; if not, the interpreter selects another element from C and continues as outlined above. Besides the data movement and the addition, the APL statement requires checking for completion and selecting data elements from the arrays. All

implementations of APL will probably perform these operations more often than any other.

The total number of passes required for executing the statement can be computed at the beginning of the execution. In the example ignore the possibility that either B or C is a multi-element array while the other is a scalar or single-element array; the number of data elements in B or C is the required number of passes through the computation described above. The passes through the computation can be counted and this number compared with the total number of passes required. The operations used in this check are some of the simplest in digital computation. These or similar operations would be available in macroinstructions as well as microinstructions, if there are any, of any general-purpose digital computer. The operations would also be easy to include in a special-purpose hardware unit to execute APL.

2.1 ADDRESS-SEQUENCING TECHNIQUES

The other operation performed many times for the execution of an APL statement is data-element selection. Given the spectrum of solid-state components likely to be available in the next five to ten years, the data elements will be stored in random-access memories. Data selection in random-access memories is accomplished by means of an address, a number that uniquely identifies a cell in the memory. To step through the data elements in an APL array

in a particular order, the elements must be stored in a random-access memory in such a way that there is an algorithm to produce a sequence of addresses that, applied to the memory, yields the desired sequence of data elements. Since one iteration of this algorithm is executed each time a data element is fetched, the time required for the algorithm to generate one address from the previous one should be as short as possible.

There are two broad classes of implementation techniques for the address-sequencing algorithm -- arithmetic and non-arithmetic.

The arithmetic technique uses the operators +, -, *, and / in the sequencing algorithm. The expression that yields the location of an element of a FORTRAN array from its subscripts and dimensions is:

$$\begin{aligned} \mathrm{LOC}(A(i_1,i_2,i_3)) &= \mathrm{LOC}(A(1,1,1)) + i_1 - 1 \\ &\quad + k_1(i_2 - 1 + k_2(i_3 - 1)) \\ &= \mathrm{LOC}(A(1,1,1)) + \sum_{j=1}^{3} (i_j - 1) \prod_{\ell=1}^{j-1} k_\ell \end{aligned}$$

LOC(X) is the location of X and A is dimensioned $A(k_1,k_2,k_3)$. The code generated by some compilers can reference array elements with less work than is required to evaluate this expression; one method can reference an array element with no multiplications. It is not necessary to subtract one from all the subscripts if LOC(A(0,0,0)) is

substituted for LOC(A(1,1,1)) in the expressions above; while A(0,0,0) does not exist in standard FORTRAN, this imaginary location simplifies the address-generation expression:

$$\begin{aligned} LOC(A(i_1,i_2,i_3)) &= LOC(A(0,0,0)) + i_1 + k_1(i_2+k_2 i_3) \\ &= LOC(A(0,0,0)) + \sum_{j=1}^{3} \left(i_j \prod_{\ell=1}^{j-1} k_\ell\right) \end{aligned}$$

The evaluation of the first of these two simplified expressions for an N-dimensional array requires N-1 multiplications and N additions. For A←B+C, this is a tremendous amount of overhead to insert at each step. But the FORTRAN address-generation expression produces an address given the subscripts of the desired element; the address-generation expression for the simple APL statement must sequence through all the addresses and need not be able to select elements in arbitrary order with the same ease. An optimizing FORTRAN compiler would use a more efficient method than the expression given above to access elements of an array, especially if the references occurred within the range of a DO loop where the index variable of the DO loop was one of the subscripts of the array reference.

The FORTRAN address-generation expression suggests a question -- are the basic arithmetic operations equally costly and therefore equally acceptable in an algorithm that should be very fast?

In the number representation common in digital computational units today, two's complement representation, addition and subtraction are equivalent. Using circuits from one of the most popular logic families (the SN74S181 and SN74S182 from the Schottky clamped transistor-transistor logic family [TI 1973]), it is possible to add two sixteen-bit numbers in about twenty nanoseconds. (A nanosecond, ns, is a billionth of a second.) This computation requires five integrated circuits -- four arithmetic logic units and one look-ahead carry-generate unit.

There is no standard way to do multiplication or division. A new integrated circuit makes it possible to do a serial (bit by bit) sixteen-bit two's complement multiplication in 800 ns with two chips (Am 25LS14 [AMD 1]). But direct comparison of the addition time with this time is not completely fair. First the addition circuit described takes as input two sixteen-bit numbers in parallel (all the bits at the same time on separate wires) and produces as output a sixteen-bit sum plus a carry, all in parallel; probably any computer system that uses an adder as one of its components would provide the inputs and use the output in parallel. The multiplier discussed above would require four to six additional integrated circuits plus some number of control circuits to convert the input from parallel to serial and the output from serial to parallel. A second objection to direct comparison of the compute times of the

adder and the multiplier is that they use circuits from different logic families with different inherent speeds. The adder uses Schottky clamped transistor-transistor logic but the multiplier uses a low-power version of that logic. The low-power family is about twice as slow as its relative [AMD 1]. But to imagine that the multiplier could be speeded up by a factor of two by changing the logic family is an oversimplification. One of the most important of the many factors limiting the amount of circuitry that one integrated circuit can carry is the power dissipation of the chip. Generally as the speed of a circuit increases, its power dissipation increases; therefore, the faster the logic used in an integrated circuit, the less of that logic can be placed on one chip. For this reason it may be impossible, with current fabrication techniques, to use the faster of the two logic families to implement the multiplier integrated circuit.

Naturally there are many ways to implement the multiplier. Another technique uses 32 integrated circuits from the same logic family as the circuits in the adder and requires about 150 ns to generate a 32-bit two's complement product from two sixteen-bit inputs (using Am25S05 integrated circuits [AMD 1]). Still another implementation requires roughly twice as many integrated circuits and can multiply sixteen-bit numbers in less than 120 ns (using SN74S274 and SN74S275 [TI 1974]). Some of the other techniques are in principle quite fast; for instance, it is

theoretically possible to treat the multiplication problem as a 32-input, 32-output Boolean function. Clearly, however, this implementation technique is far beyond the bounds of practicality. It is possible to conclude that, given the selection of solid-state digital logic components likely to be available in the next five years, addition and subtraction circuits are simpler and faster than multiplication circuits of the same precision.

The three implementations discussed above used integrated circuits designed especially for multiplication; the logic family of these special integrated circuits, which is the family with the broadest selection of functions available today, includes no integrated circuits intended specifically to aid the implementation of division. As a result, a well designed division implementation will be slower or more complex than a comparably well-designed multiplication implementation.

An address-generation scheme based on addition or subtraction would be simpler to implement and would run faster than a scheme based on multiplication or division. The discussion said nothing about the relative flexibility or power of the two approaches.

It is also possible to use non-arithmetic operations in an address-generation algorithm. The reason one might consider this is that the Boolean functions are faster than addition; they are bit-by-bit operations, and unlike

addition, they do not require carry propagation. There are linear and non-linear sequence generators, both of which are shift registers with some type of feedback. Linear sequence generators use EXCLUSIVE-OR functions (addition modulo two) in the feedback network, and non-linear sequence generators use other Boolean functions. Because references to a specific data element in an array must be handled efficiently, it must be easy to generate any member of the address sequence, and, of course, the sequences must be of variable length. These two requirements argue against the use of sequence generators; furthermore, even the simplest sequence generator, which is certainly not flexible enough for address generation, can generate addresses only slightly faster than an adder -- about 15 ns per number. The fastest high-density random-access memory in this family has an access time of about 30 ns (93415A [Fairchild 1975]). Components must be added to the memory circuits to make a memory system and there must be some allowance for speed variations, so 60 to 100 ns is a more reasonable access time for a memory system made from these fast memory components. The additional five nanoseconds, even if it were possible to gain them, are insignificant.

The net result is that an address-sequencing algorithm should, if possible, be based on addition.

2.2 LADDERS

A ladder is a structure, introduced by Perlis [1975], that combines a data-access mechanism for array-structured data with computational and linkage capability. One ladder is associated with each occurance of an array in an APL statement. The ladders for the arrays referenced by a statement are connected to form a ladder network for the statement.

A ladder can be divided into two parts, a fixed part and a variable part. The fixed part handles the accessing of the array elements and the check for completion. It is determined by the rank of the array with which it is associated. The variable part performs the computation specified by the operators in the statement.

The fixed framework ladders provide allows code for an APL statement to be generated easily. The fixed parts of the ladders for an APL statement can be formed as soon as the ranks of the arrays referenced by the statement are known. If the ranks are not known at the time the statement is to be executed then the statement must be broken into parts. The first part is the largest part, beginning at the right of the statement, for which the ranks of all the arrays are known. The ladder network for this statement fragment can be formed and executed. After the execution, another ladder network is formed from the rest of the statement in the same way as before. Then the new network

is executed. This process continues until the entire statement has been executed. Any legal APL statement can be executed in this manner.

The variable parts of each ladder are generated in much the same way code is generated by a standard compiler. If the ladder network for a statement is saved after the statement has been executed, it is likely that the same network can be used the next time the statement is encountered. The conditions in which a ladder network can be reused are discussed in more detail below.

Ladders can be explained most effectively by first discussing the workings of the data-access mechanism. The mechanism used is similar to others [Abrams 1970, Hassitt 1972].

Let A be an array of rank three and shape k_1, k_2, k_3 stored in ravel (row-major) order. The separation between adjacent elements in the same row (differing in only the last subscript) is 1. The separation between elements differing only in the second subscript (and there only by one) is k_3, or the number of elements in a row. And the separation between elements differing by one in the first subscript and not at all in the others is $k_2 \times k_3$, or the number of elements in a planar cross-section of the array. This storage arrangement is the same as the one used for storage of FORTRAN arrays except that the order of the subscripts is reversed. The analogous expression that gives

the location of a particular element of the array is:

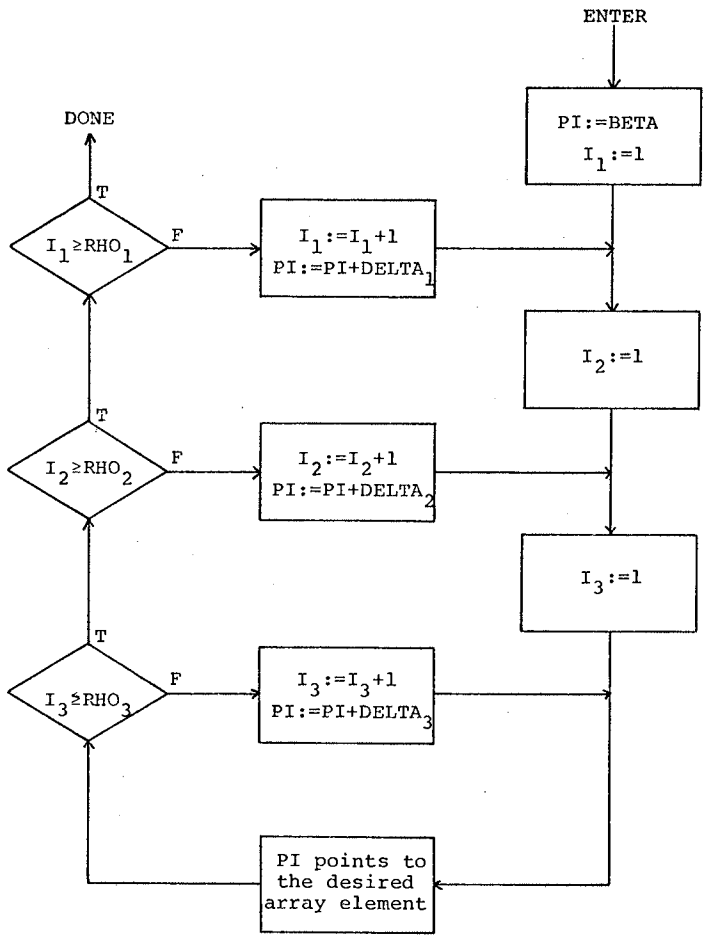
$$\begin{aligned} LOC(A[i_1;i_2;i_3]) &= LOC(A[1;1;1]) + i_3 - 1 \\ &\quad + k_3(i_2-1) + k_2k_3(i_1-1) \\ &= LOC(A[1;1;1]) \\ &\quad + \sum_{j=1}^{3} [(i_j-1) \prod_{\ell=j+1}^{3} k_\ell] \end{aligned}$$

Consider the problem of delivering, in ravel order, the elements of the array A to a computational procedure. The address of the element to be delivered must be initialized to the address of the first element. Since the array is stored in row-major order, the address of the next element in the row may be produced by adding one to the current address. Continuing to apply this procedure will yield the sequence of addresses of all the data elements in the first row. Because rows are stored sequentially, the last element of one row is followed immediately by the first element of the next row; therefore, adding one to the address of the last element of one row will generate the address of the first element in the next. In fact, adding one will always produce the address of one data element from the address of the previous data element. This procedure obviously works; it is clear that beginning with the address of the first element of an array stored in ravel order and adding one to that address until the address of the last element in the array is reached will produce the sequence of addresses of the data elements in ravel order. A simple extension of

this technique can produce the sequence of addresses of the elements of an array in ravel order even though the array itself is not stored in ravel order.

The mechanism depicted in Figure 2-1 is designed to produce the addresses of the elements of an array in ravel order. In the figure, PI is a pointer that is initialized and stepped so that it sequences through the elements in the array referenced by the ladder in ravel order. BETA represents the address of the first element in the array, the array's base address. The RHO variables are the components of the shape or dimension vectors of the array and the I variables are the subscripts. The origin is assumed to be one. When control passes through the bottom block, PI points to the array element specified by the subscripts.

It is clear that the mechanism will step through the elements of an array stored in ravel order ($DELTA_1 = DELTA_2 = DELTA_3 = 1$). Appendix B shows that the application of certain common APL operators to an array stored to permit access in ravel order by this mechanism yields an array that can be accessed in ravel order by the same mechanism but with different BETA, RHO, and DELTA. Since these operators change the logical order but not the physical order of the data, the resulting array is not stored in ravel order. The mechanism can be used to access these arrays, so storage orderings that the mechanism can handle and that differ from



Data-access mechanism

Figure 2-1

the ravel ordering are common. The operators, which Abrams called selection operators, are reverse, transpose, take, drop, and certain types of subscripting.

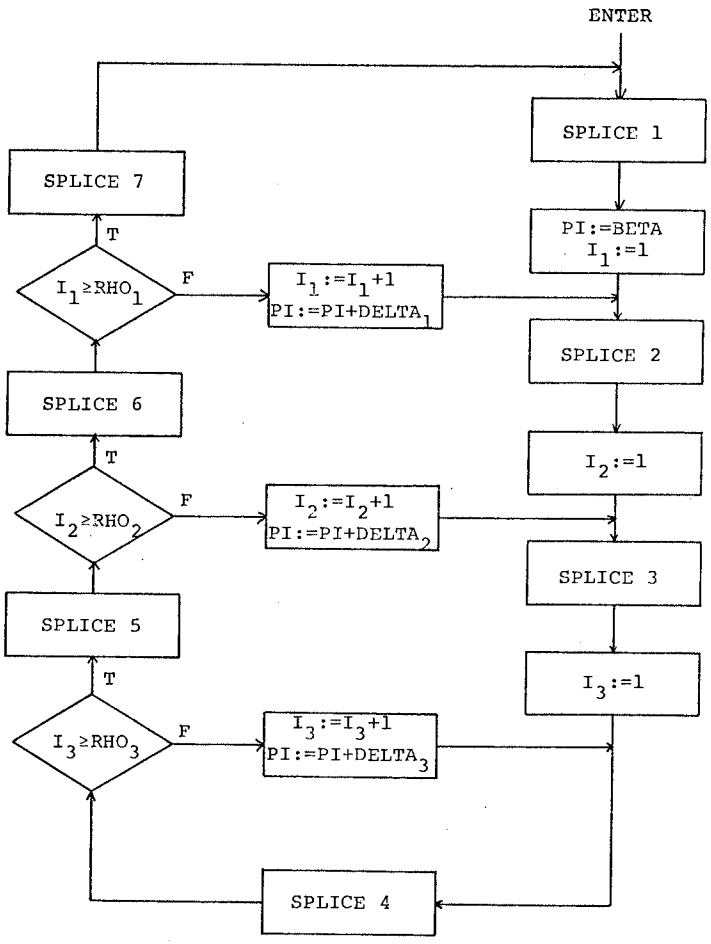
It is sometimes necessary to access a particular element of an array, as specified by its subscripts. For an array that the procedure described above can access, Appendix A shows how. The address of a particular element of an array is given by:

$$LOC(A[j_1;j_2;\ldots j_N]) = BETA + \sum_{i=1}^{N} (j_i - 1)G_i$$

The G_i are defined in Appendix A.

Not only does the mechanism sequence through the addresses and perform a check for completion, but it also enables some computations to be simplified. The question remains how to interface this structure with the computation still required by the statement.

Figure 2-2 shows a ladder structure for an array of rank three as described above except that seven numbered boxes, called splices, have been inserted in the structure. Code to perform the calculations associated with the statement is placed in the splices. For an APL statement involving only the simple scalar operators, all code goes into the splice in the innermost loop, splice 4 for the rank-three case. In addition one of the arrays has an END instruction in the splice encountered only when an entire



Ladder mechanism

Figure 2-2

pass has been made through the array, splice 7 for the rank-three example. The additional splices are used for the more complicated operators like outer and inner product, scan, and reduction. The instructions in the splices are simple scalar-oriented instructions; they include the basic arithmetic operations, some of the Boolean operations, and testing and branching instructions.

The ladder structure provides a way to step through the elements in an array and to perform computations on them. Since almost all APL statements involve more than one array, there must be a way to link ladders so that data as well as control can be transferred among them. The data can be transferred by allowing the code in each ladder to reference a shared memory where all data (except the arrays themselves) are stored. The binding of the code to this shared data takes place at code-generation time. Control is passed among ladders by means of coroutine jumps.

Most operators in APL can be accommodated within the computational structure outlined above. The fixed address sequencing that the ladder mechanism provides is not suitable for some APL operators such as grade up and grade down; such operators can be handled by standard interpretation techniques.

The ladder structure handles changes in the shape of arrays automatically. Once the ladders and the associated splice code have been generated, they can be used the next

time the statement is executed if the conditions described by Perlis [1975] are not violated. The rank and type of each variable in the statement may not change from execution to execution; every function called in the statement must return a result with the same rank and type for every call provided the arguments also have the same rank and type; and no (X)⍴Q or (X)⍉Q may occur where X is an expression other than a constant vector and Q is any expression. Of course, a perfectly legal APL program can violate these conditions, and, as a result, the ladders and splice code will have to be regenerated.

These limits on the use of ladders in the execution of APL programs detract from the elegance of the approach, but the thrust of this work is to improve the overall efficiency of execution of APL programs even at the expense of some elegance.

In summary, ladders provide a data-access mechanism for arrays in APL statements. Some APL operators can be implemented by altering data-access parameters without fetching any elements of the array argument; most others can be executed by inserting code into the splices in the structure. For most APL statements the ladders and splice code generated when each statement is first executed can be reused for subsequent executions, thus reducing the interpretation/compilation overhead. A normal interpretation scheme can be used for those APL statements

that contain operands or operators not easily implemented with ladders, such as null arrays and grade up/grade down. Since, for many APL statements, accessing array elements requires a large fraction of the statement's execution time, an efficient accessing mechanism will allow the construction of a fast APL processor.

CHAPTER 3

LADDER IMPLEMENTATION

The mechanism discussed in Chapter 2 used a number of parameters in controlling the address-sequence generation. The data-access pointer, called PI, the base address, called BETA, and the rank of the array are all parameters used in the mechanism. The variable PI contains the current address in the address sequence and BETA is the initial value of this sequence. The rank is used to generate a ladder structure of the correct depth for an array. How many other parameters are necessary depends on the rank of the array because the remaining parameters have a different element for each coordinate position. The vector of subscripts of the element whose address is in PI is maintained and used with the shape vector, which is also stored, to control the number of passes through each loop in the ladder structure. The vectors RHO and G are also stored. As discussed in Appendix A, G can be computed from DELTA, RHO, and the rank; similarly DELTA can be computed from G, RHO, and the rank, so it is not necessary to store them both. Since a reference is made to an element of the DELTA vector on each step of the address generation, this vector should certainly be stored. The G vector is used to calculate the address of an array element from its subscripts and, as Appendix B shows, to handle the selection operators. Storing both DELTA and G is particularly handy if a specially designed piece of hardware is used to implement the ladder structure

because the number of items stored per coordinate is four instead of three. The address of a data-access parameter can be formed by shifting and masking rather than by adding and multiplying.

In addition to these parameters, the type of an array must be stored to allow the address-sequencing mechanism to perform type checks at run time. For some ladder implementations the type is used in address sequencing because data elements of different types require different amounts of storage and the step between two elements in an array will depend on the type of the array.

A ladder uses these parameters to sequence through the addresses of the elements in an array. The execution of an APL statement requires adding code (called splice code) to the ladder structure and passing data and control between the ladders used for the statement (generally one ladder for each occurrence of an array in the statement). Besides the code used to perform the calculations explicitly specified by the statement, each ladder has code to perform the address-sequencing calculations. With each ladder there are two types of data -- the data-access parameters and temporaries (data both from the array and from calculations on the array data) -- and two types of code, the code that controls the address sequencing and the splice code. The code controlling the address sequencing need not reference the temporaries, but examples will be given later in which

the calculation code must reference both data types. One ladder seldom needs to reference the data-access parameters of another ladder. Although this capability could be used to include conformability checking in the ladder code, better methods are discussed below.

The number of storage locations that the data-access parameters for a single ladder need is modest, roughly six plus four times the rank of the array accessed by the ladder. A study of over 8500 lines of "high quality" APL done by Saal and Weiss [1975] showed that, where the rank of an array could be determined syntactically (from a subscript expression), less than one per cent of the array references had a rank of three and none had a rank of four or greater. If we place an upper limit of six on the rank of an array that can be accessed using the ladder structure, then no more than 32 storage locations are required for the data-access parameters (four parameters each for six coordinates, plus up to eight parameters not duplicated for each coordinate).

The number of storage locations ladders use for the temporaries is also small. Array elements are brought into this storage and intermediate results of the splice code are stored here. Since all ladders use the same temporary memory, the splice code in one ladder has access to the results of splice-code calculations done in other ladders. A simple scalar operator requires one storage location to

store its result; some of the more complicated operators (for instance, decode and encode) require two or three storage locations. The number of temporary locations that a ladder network for an APL statement needs is usually smaller than the number of nodes in its parse tree. These numbers are close only when a statement contains many of the most complicated operators. For instance the statement A←B+C has five elements in its parse tree, yet it requires only one temporary. Saal and Weiss give a histogram of the number of elements in the parse tree per line. The maximum number of elements is 35, and the average is nine. A statement with twenty elements in its parse tree almost certainly contains fewer than fifteen operators. Since most of the operators require one temporary and those that require more require only two or three, 32 temporaries are sufficient for most APL statements. In the special hardware designs discussed in Chapter 4, many more temporaries are included to allow even very complex statements to be executed without being divided into smaller statements.

If we assume that an average statement analyzed by Saal and Weiss, which has nine elements in its parse tree, contains five array references and four operators, then temporaries and data-access parameters require less than 200 locations of storage (32 temporaries plus 32 data-access parameters for each of the five arrays). The amount of code required depends on the instruction set used to implement the code. If the instruction set of a typical computer is

used, the number of instructions necessary to implement the address generation for an array is about five times the rank of that array. (See the example in the software discussion below.) If the four operators in the statement are simple, they take ten to fifteen splice-code instructions; for more complicated operators the code might expand by a factor of two or three. The total storage requirement for the code for an APL statement with five array references and four operators is probably less than 100 locations -- fifty locations for the ladder code for five arrays of rank two, plus fewer than fifty locations for the splice code for four operators.

Thus the total storage requirement for an average APL statement is less than 300 locations; the storage requirement is this large only when the statement is active (being executed). This requirement does not count the storage for the elements of the arrays in the statement. The only information, besides the statement itself, that must be maintained when the statement is not active is the rank, shape, base address, type, and the G or DELTA vectors for each array. A standard APL interpreter must maintain all this information except for the G or DELTA vectors. In some cases it is possible to reuse the splice code (see Chapter 2), so a ladder-based APL system would probably save this as well. The inactive storage space can be decreased at a cost of added regeneration time and vice versa.

3.1 SOFTWARE

The first of the implementations is based on software. The detailed examples in this section will use the Digital Equipment Corporation PDP-11 computer [1975], although any general-purpose digital computer would suffice.

If we implement ladder structures on a general-purpose digital computer, all the storage used for data-access parameters, temporaries, array elements, splice code, and address-generation code will be the same -- the main memory of the machine. During the execution of any one statement almost every location referenced will be one of a few hundred, but this is of no direct benefit. On computer systems with a cache memory this locality can be of tremendous benefit depending on the interaction of the assignment of code and data to locations in memory and the cache organization and replacement algorithm. But on most systems the cache is organized to optimize average behavior. It is difficult for a naive programmer to write a program that uses the cache poorly; it is also difficult for a knowledgeable programmer to exploit the cache. And on a multiprogramming system, the cache is almost completely insulated from the programmer.

In a software implementation it is impossible to take advantage of the repetition of a sequence of instructions. The code that performs the address sequencing in the innermost loop is very simple, but it will probably require several instructions to implement on a general-purpose computer. The address-sequencing code references fewer than 100 locations (see Figure 3-1 below) and uses only direct addressing, but a programmer cannot take advantage of this simplicity.

The following example illustrates the applicability of the ladder mechanism to a software implementation of APL. Consider the statement A←B+C. When the APL processor encounters this statement it must first check to see that the statement can be executed -- that B and C are defined and conformable and that there is enough free storage to contain A. If the statement is executable, the processor must create a ladder network; it might begin by creating the address-sequencing mechanisms.

The code in Figure 3-1 illustrates one way to do address sequencing for an array of rank three. The numbered lines represent the splices or places to insert code. A copy of this code block must be created for each occurrence of each array in the statement; address-sequencing code blocks for an array are completely determined by the rank of the array. Each occurrence of an array in the APL statement has its own PI, I1, I2, and so on. After creating the address-sequencing code blocks, the APL processor scans the APL statement from right to left and fills in the splice code. For this simple example, splice 4 of the ladder for C is loaded with code that loads a temporary with the element

```
LP0:    1               ;SPLICE 1
        MOV BETA,PI     ;INITIALIZE DATA-ACCESS
                        ;POINTER
        MOV #1,I1       ;INITIALIZE FIRST
                        ;SUBSCRIPT
LP1:    2               ;SPLICE 2
        MOV #1,I2       ;INITIALIZE SECOND
                        ;SUBSCRIPT
LP2:    3               ;SPLICE 3
        MOV #1,I3       ;INITIALIZE THIRD
                        ;SUBSCRIPT
LP3:    4               ;SPLICE 4
        CMP I3,RHO3     ;IF THE SUBSCRIPT IS
                        ;GREATER THAN OR EQUAL TO
        BGE OVR3        ;THE CORRESPONDING ELEMENT
                        ;OF THE SHAPE VECTOR
                        ;THEN MOVE TO THE NEXT
                        ;OUTER LOOP
        INC I3          ;ELSE STEP SUBSCRIPT
        ADD DELTA3,PI   ;AND DATA-ACCESS POINTER
        BR LP3          ;AND LOOP
OVR3:   5               ;SPLICE 5
        CMP I2,RHO2     ;IF THE SUBSCRIPT IS
                        ;GREATER THAN OR EQUAL TO
        BGE OVR2        ;THE CORRESPONDING ELEMENT
                        ;OF THE SHAPE VECTOR
                        ;THEN MOVE TO THE NEXT
                        ;OUTER LOOP
        INC I2          ;ELSE STEP SUBSCRIPT
        ADD DELTA2,PI   ;AND DATA-ACCESS POINTER
        BR LP2          ;AND LOOP
OVR2:   6               ;SPLICE 6
        CMP I1,RHO1     ;IF THE SUBSCRIPT IS
                        ;GREATER THAN OR EQUAL TO
        BGE OVR1        ;THE CORRESPONDING ELEMENT
                        ;OF THE SHAPE VECTOR
                        ;THEN MOVE TO THE NEXT
                        ;OUTER LOOP
        INC I1          ;ELSE STEP SUBSCRIPT
        ADD DELTA1,PI   ;AND DATA ACCESS POINTER
        BR LP1          ;AND LOOP
OVR1:   7               ;SPLICE 7
        BR LP0          ;CYCLE THROUGH THE
                        ;ENTIRE ARRAY AGAIN
```

Address-Sequencing Code For Rank Three Array

Figure 3-1

that C's PI points to: MOV @PIC,T1. Splice 4 of the ladder for B is loaded with ADD @PIB,T1. Note that the PI variables referenced in these two statements are different, one for the C ladder and one for the B ladder, but that the variable T1 referenced by both statements is the same. Next a MOV T1,@PIA instruction is inserted into splice 4 of the ladder for A. These three instructions perform all the calculations in the example but the APL processor must provide some means for transferring control among the ladder structures. The mechanism for transferring control discussed in Chapter 2 is the coroutine jump. It is possible to simulate a control transfer between two coroutines using the subroutine jump instruction on the PDP-11, but this technique can't be extended beyond two coroutines. One technique to implement a coroutine jump is illustrated by the three code segments in Figure 3-2. In the figure, T3 and T5 have been initialized to point to coroutines 2 and 1. In this example control passes from coroutine 3 to coroutine 2, to 1, back to 2, and finally back to 1. This code for a coroutine jump is not position independent, but it can be relocated on a system with hardware relocation capability.

The code blocks for the statement A+B+C with the simulated coroutine jumps in place are shown in Figures 3-3, 3-4, and 3-5. This is a lot of code, but generally only the code in the innermost loop affects the execution time of the statement. In this example the innermost loop has 26

```
        MOV #RET1,T5
        JMP @T4
RET1:

        Coroutine 1



        MOV #RET2A,T4
        JMP @T5
RET2A:
        .
        .
        .
        MOV #RET2B,T3
        JMP @T2
RET2B:

        Coroutine 2



        MOV #RET3,T2
        JMP @T3
RET3:

        Coroutine 3
```

Coroutine Linkage

Figure 3-2


```
LP0A:   MOV BETAA,PIA
        MOV #1,I1A
LP1A:   MOV #1,I2A
LP2A:   MOV #1,I3A
LP3A:   MOV T1,@PIA
        MOV #RETA,T5
        JMP @T4
RETA:   CMP I3A,RHO3A
        BGE OVR3A
        INC I3A
        ADD DELTA3A,PIA
        BR  LP3A
OVR3A:  CMP I2A,RHO2A
        BGE OVR2A
        INC I2A
        ADD DELTA2A,PIA
        BR  LP2A
OVR2A:  CMP I1A,RHO1A
        BGE OVR1A
        INC I1A
        ADD DELTA1A,PIA
        BR  LP1A
OVR1A:  BR  LP0
```

Ladder for Array A

Figure 3-3

```
LP0B:   MOV BETAB,PIB
        MOV #1,I1B
LP1B:   MOV #1,I2B
LP2B:   MOV #1,I3B
LP3B:   ADD @PIB,T1
        MOV #RETBA,T4
        JMP @T5
RETBA:  MOV #RETBB,T3
        JMP @T2
RETBB:  CMP I3B,RHO3B
        BGE OVR3B
        INC I3B
        ADD DELTA3B,PIB
        BR  LP3B
OVR3B:  CMP I2B,RHO2B
        BGE OVR2B
        INC I2B
        ADD DELTA2B,PIB
        BR  LP2B
OVR2B:  CMP I1B,RHO1B
        BGE OVR1B
        INC I1B
        ADD DELTA1B,PIB
        BR  LP1B
OVR1B:  BR  LP0B
```

Ladder for Array B

Figure 3-4


```
LP0C:   MOV BETAC,PIC
        MOV #1,I1C
LP1C:   MOV #1,I2C
LP2C:   MOV #1,I3C
LP3C:   MOV @PIC,T1
        MOV #RETC,T2
        JMP @T3
RETC:   CMP I3C,RHO3C
        BGE OVR3C
        INC I3C
        ADD DELTA3C,PIC
        BR  LP3C
OVR3C:  CMP I2C,RHO2C
        BGE OVR2C
        INC I2C
        ADD DELTA2C,PIC
        BR  LP2C
OVR2C:  CMP I1C,RHO1C
        BGE OVR1C
        INC I1C
        ADD DELTA1C,PIC
        BR  LP1C
OVR1C:  BR  LP0C
```

Ladder for Array C

Figure 3-5

instructions -- eight in the code for array C, ten in the code for array B, and eight in the code for array A. Of these 26 instructions, three deal directly with array data, eight transfer control among the code blocks by simulating coroutine jumps, and the other fifteen provide address sequencing, subscript maintenance, and completion checking. If both B and C are stored in ravel order, the blocks of code in Figures 3-6a and 3-6b produce the same result as the code in Figures 3-3, 3-4, and 3-5.

Three of the four instructions in the loop of the code in Figure 3-6b correspond to the three instructions dealing with array data in the ladder code above. In the non-ladder code the splices have been merged into one code segment so that no coroutine jump instructions are needed. The non-ladder code segments do not maintain subscript values as they step through the arrays. The code that fetches the array data also steps the address by using the PDP-11's auto-increment addressing mode. In this mode a general-purpose register provides the address of the data item to fetch, and afterwards the register is stepped to point to the next sequential location. If the data items were not adjacent, three instructions to do address sequencing would have to be added to the loop of the code segment. In each case one instruction is required to check for completion. Almost all of the code saving in the loop of the non-ladder example is due to eliminating the overhead imposed by the ladder structure.


```
        MOV RHO3C,R0
        MUL RHO2C,R0
        MUL RHO1C,R0
        MOV BETAC,R1
        MOV BETAB,R2
        MOV BETAA,R3
LOOP:   MOV (R1)+,(R3)
        ADD (R2)+,(R3)+
        SOB R0,LOOP
```

Assembly Code for A←B+C

Figure 3-6a

```
        MOV RHO3C,R0
        MUL RHO2C,R0
        MUL RHO1C,R0
        MOV BETAC,R1
        MOV BETAB,R2
        MOV BETAA,R3
LOOP:   MOV (R1)+,R4
        ADD (R2)+,R4
        MOV R4,(R3)+
        SOB R0,LOOP
```

Assembly Code for A←B+C

Figure 3-6b

Cross-machine comparisons of performance are difficult, but Breed and Lathwell [1967] give performance figures that allow comparing their APL interpreter running on an IBM 360/50 with a software implementation of APL based on ladders. They say that an APL statement will often run between a fifth and a tenth as fast as the compiled version of the statement. The code for the simple APL statement above had 26 instructions in the innermost loop; the optimum assembly-language implementation of the same statement had three. If we assume that instruction count and execution time are approximately proportional, then the ratio of the execution time of an APL statement to the execution time of the optimum equivalent assembly code is about the same for the IBM 360/50 and for the DEC PDP-11. This suggests that a software implementation of APL using ladders is comparable to the Breed implementation.

This conclusion raises two questions -- what, if any, are the advantages of using a ladder-based software implementation of APL over a more standard interpreter, and what are the advantages over the more direct implementation discussed for the PDP-11?

First, as Appendix B describes, ladder structures allow very efficient implementation of the selection operators. Bingham [1975] found in an analysis of APL programs that about ten percent of all operators used in the 10,000 lines of APL code he examined were take, drop, reverse, or

transpose, and that about 25 per cent of the array references were subscripted. Unfortunately, Bingham does not analyze the subscript expressions, but it is reasonable to assume that many of the subscript expressions can be handled directly with the ladder mechanism. A software ladder-based APL system would probably be a little larger than a standard APL interpreter. The time savings that would result from the direct execution of the selection operators is not large enough to be a conclusive argument for a ladder-based software APL implementation.

There are advantages to using ladders in a software APL implementation. A ladder network computes the result of an APL expression one element at a time; it finishes computing one element of the result before beginning the next. This greatly reduces the amount of temporary storage required in evaluating an expression. Savings are most pronounced when an outer product or a multi-dimensional inner product is followed by a reduction or a rank-reducing transpose. The sequence of operators outer product followed by reduction is particularly useful.

For example the APL expression (2=+⌿0=(⍳N)∘.|⍳N)/⍳N computes the primes between one and N. The largest value of N for which this expression can be evaluated depends on the amount of storage the APL system can use. For the APLSS system on the PDP-10 of the Yale University Department of Computer Science the maximum value of N is about 100. The

set of programs written to simulate a ladder-based APL system (see Appendices E and F) can evaluate the same expression, although much more slowly, for larger values of N. The value of N is limited by the space available to store the primes output by the expression. If the primes need not be stored there is no limit on the value of N imposed by storage availability. Of course, the time required to evaluate the expression rapidly becomes prohibitively large as N increases, and eventually N can no longer be represented in a single word of storage. But there is a large range of N for which a ladder-based APL system can evaluate the expression and a standard APL interpreter cannot.

In Bingham's study, inner product and outer product made up about 1.5 per cent of the operators, and in Saal and Weiss's study the same operators made up about .8 per cent. These operators might be used more if they were not so costly in storage. These are arguments for using a ladder-based software APL system rather than a standard software APL interpreter, but it is not clear that these arguments are strong enough to justify the increased size of the software APL processor.

Similar comparisons can be made between a ladder-based software implementation and a more compiled implementation. If the arrays must be stored in ravel order then none of the techniques discussed in Appendix B can be used to perform

the selection operations. There are a number of difficulties in generating code for the more complicated APL operators. Networking of highly compiled code is more difficult than it is for ladder code. Nevertheless, very significant gains in efficiency are possible with simple operators and expressions, and the studies by both Bingham and Saal and Weiss show that these operators and expressions are the most common. Generating highly optimized code for simple expressions and less optimized code (using ideas like the ladder mechanism and ladder networks) for more complicated expressions might be a good way to execute APL with software alone. There is a great deal of room for further work in this area.

3.2 FIRMWARE

It is also possible to use firmware to implement an APL processor based on ladders; firmware has been used to implement other APL processors [Zaks 1971, Hassitt 1973, Grant 1974, Hassitt 1975]. Generally firmware refers to all types of microprograms, but here it will refer to microprograms written for computers with a writable control store. This implementation technique is not applicable to all computers; neither the PDP-10 or the PDP-11 is suitable. The KL-10, a new model of the PDP-10, is microprogrammable and has a writable control store, but the manufacturer advises users not to microprogram it themselves. Almost all of the PDP-11 family of processors

are microprogrammed but none has a writable control store. A group at Carnegie-Mellon University extended the design of the PDP-11/40 to include a writable control store and more processing capability [Oakley 1975]; the applicability of this machine, called the 11/40E, will be discussed briefly below.

Judging a firmware approach to an APL processor is harder than judging software because microinstruction sets and the organizations of the machines they control vary more than macroinstruction sets and high-level machine organization do.

There are two classes of microprogrammed machines -- those with horizontal microcode and those with vertical microcode. This distinction concerns the level at which the components of the machine can be controlled.

Vertical microcode tends to be narrow; the IBM 360/25 and the Burroughs B1700 have microinstructions that are sixteen bits wide [Bell 1971, Tanenbaum 1976]. These instructions usually resemble simple macroinstructions such as ADD, MOVE, AND, and so on. The possible source and destination operands of these instructions include the registers accessible at the macroinstruction level and a number of special registers (for instance memory address and memory data registers). Vertical microinstructions usually do not have timing sensitivities and they often obscure the detailed organization of the machine on which they run.

Because the control of several independent functions is sometimes encoded into a single field of a vertical microinstruction, some operations that the underlying hardware can do are precluded by the structure of the microinstruction-decoding logic. Some kinds of parallelism are lost, and with them, some efficiency. Because of their simplicity, vertical microprograms are often easy to write. In fact a person with some assembly-language programming experience but no knowledge of hardware can probably read and understand most vertical microprograms.

Horizontal microcode is usually wide -- the IBM 360/50 and the nanolevel of the Nanodata QM-1 use a ninety-bit microcode [Bell 1971, Nanodata], the 11/40E uses an eighty-bit code [Oakley 1975], the 11/45 uses a 64-bit code [DEC 1972], and the 11/40 uses a 56-bit code [Tanenbaum 1976]. Horizontal microcode is usually written in a format different from assembly language. A horizontal microinstruction has a number of fields, each controlling a component of the machine. The fields specify such things as the arithmetic logic unit (ALU) function, the source of the two inputs to the ALU, which registers are gated onto the data paths in the machine, which registers are loaded with new data, and the source of the new data. Horizontal microinstructions are as varied as the organizations of the machines on which they run, and machines at this level of computer architecture vary more than at the level where macroinstructions specify the actions of the machine.

Microprogramming a machine that uses a horizontal microcode requires intimate knowledge of the internal structure and timing of the machine. Even then it is more difficult to code an algorithm using a horizontal microinstruction set than a vertical one. There are microcode assemblers for some machines with horizontal microcode, but the input to these programs is foreign to anyone unfamiliar with the machine's architecture.

Horizontal microinstructions, which get at the full power and flexibility of the underlying hardware, often result in substantial parallelism in microprograms. For this reason horizontally coded machines present greater potential than vertically coded machines for implementing a ladder-based APL processor efficiently. In a machine with a horizontal microinstruction set, the power is limited only by the structure. In a machine with a vertical microinstruction set, there are operations that the machine can perform but the programmer cannot specify.

In order to examine microprogrammable machines, we need to review the resources used by a ladder network. First, a complicated APL statement requires a few hundred storage locations for data-access parameters and temporaries. (Ten arrays of rank four connected with operators that are hard to implement require about 200 locations.) Fewer than 100 storage locations are required for the splice code for a similar statement. The PDP-11 software implementation of

the ladder network for the statement also requires a few hundred locations for code to do address sequencing, subscript maintenance, and completion checking. The total storage requirement is less than one thousand locations. Although it is not clear how to reduce significantly the storage for the data-access parameters and temporaries and still keep the ladder mechanism, special instructions can reduce the storage the splice code and the address-sequencing code need. A coroutine jump instruction that requires a single location will reduce the storage used by the splice code. The iteration step for a single coordinate requires five PDP-11 instructions; with a custom instruction set this can be reduced to one. Since there is an iteration-step instruction for each coordinate of each array in an APL statement, the total ladder-code requirement for a statement is reduced by four times the sum of the ranks of all the arrays in the statement. A ladder network of ten arrays of rank four requires 280 PDP-11 instructions for address sequencing, subscript maintenance, and completion checking. (See examples in the software section above.) A special iteration-step instruction reduces this requirement to 120 instructions.

This discussion suggests two features of a microprogrammable computer that would be useful for the firmware implementation of a ladder-based APL processor. The first is a few hundred words of very fast memory that can be used for both instructions and data; this storage

makes it possible to eliminate all main-store references except those for array data (those that use the data-access pointer). The second is powerful field-extraction so that the computer can emulate different types of macroinstructions efficiently.

In the discussion of the software implementation, the example was based on the PDP-11; here the firmware implementation uses the 11/40E, a PDP-11/40 with extensions. (Oakley [1975] gives a brief discussion of the 11/40E and gives a reference for a more detailed description.) The designers of the PDP-11/40 selected the machine's architecture so that the PDP-11 instruction set could be implemented with it easily. The unmodified 11/40 lacks both desirable features. It has only sixteen locations of fast memory, and its field extraction and testing capability is useless for emulating instructions that differ greatly from the instructions of the PDP-11. The 11/40E has a general shift/mask circuit for field extraction, and the control memory has been modified so that it can be used for data as well as microinstructions. Because of this dual role it is unwieldy for data storage; all data should probably be stored in the main memory which is accessed via the UNIBUS. Accessing this memory to get data usually takes much longer than executing the microinstructions that use the data. As a result, the execution time of a microprogram is primarily determined by the number of main memory references it makes. The PDP-11 code for an iteration-step instruction given

above makes fifteen references to main memory when it is executed and the coroutine-jump code makes four. A firmware implementation of these two operations of the 11/40E requires less than half the number of main memory references. The firmware-assisted operations should take about half the time required by their software equivalents. Since, in the simple example covered while discussing software ladder processing, these overhead functions accounted for 23 of the 26 instructions in the innermost loop, the overall improvement should be close to a factor of two. The highly optimized assembly code probably cannot benefit from new instructions since the normal PDP-11 instructions used are close to ideal for the example. But an APL processor based on ladders still takes longer to execute a simple APL statement than equivalent highly optimized code.

The difference in processing speeds would be further cut if there were a high-speed memory with 256 to 1024 locations for storing data-access parameters. An iteration step is made up of simple tasks; most of the time is spent waiting for the data to arrive from the main store. If this time were reduced then the time for the computations would be cut correspondingly. The array will probably always be stored in the main store, so adding a small, fast memory will not affect access time to it. This addition will help the performance of the ladder APL processor more than it helps the optimized code.

There are machines that have both features. The MLP-900, a 36-bit microprogrammable machine with a vertically encoded microinstruction set designed by the Standard Computer Corporation, is one such machine. John D. Oakley of the Department of Computer Science at Carnegie-Mellon University compared it to the 11/40E using several test problems. (Oakley [1975] also gives a discussion of the MLP-900 and references for a more detailed description.) He concluded that, on problems where a large computational path width and a large quantity (up to 1K) of fast storage are useful, the MLP-900 is up to four times faster than the 11/40E; on problems where they are not an advantage, the 11/40E is slightly faster. The sixteen-bit data path width of the PDP-11 is insufficient for floating-point numbers and is probably insufficient for integers in APL. While questions of data path width are independent of the control questions addressed here, they must be dealt with some time. In any event, the APL processor based on ladders has both characteristics that favor the MLP-900 over the 11/40E. Horizontal microcode is usually better than vertical for implementing ladders, but not here. The reason is clear -- no matter how much flexibility a horizontal microcode allows, it cannot allow features that are not in the underlying machine. Unfortunately the only MLP-900 in existence is a prototype and there are no plans to put it into production.

The final machine to consider for a firmware implementation of ladders is the Burroughs B1728, a vertically encoded microprogrammable computer with 24-bit data paths in the CPU [Burroughs 1972, Tanenbaum 1976]. The B1728 differs from other machines in the ease with which it can work with variable-size data -- both in the CPU and in memory. Some of the language processors for the B1728 use this capability by working with instructions of different length. Adapting the size of the data element to be fetched next from memory allows an immense variety in the macroinstructions a firmware processor can interpret. Using this to extract fields from an instruction forces the main memory to fetch only one field at a time -- time-consuming for an instruction with several small fields. The more usual shift/mask capability of the B1728 is faster for instruction decoding.

The processing unit of the B1728 provides only a small amount of fast memory for data. Microprograms can overflow the control memory, where they are normally, into main memory, and be executed from there. This removes the limit on the size of microprograms that can be run on a B1728. But access to the main memory is four times slower than access to the control memory [Burroughs 1972].

Like the 11/40E, the B1728 has a shift/mask unit that can extract fields from instructions and very little fast memory that can be used for data storage. The B1728

provides the additional capability, although with some decrease in interpretation speed, of interpreting macroinstructions encoded efficiently and compactly. That is, the B1728 memory-mapping unit makes possible smaller but slower ladder code. Currently no more than 128K bytes of main memory can be installed on the B1728, and this limit would be a handicap if the APL processor were part of a timesharing system.

The merit of a firmware implementation of ladders depends upon the host machine. Special firmware-assisted instructions can speed up by a factor of two a ladder-based APL system on the PDP 11/40E. Considerably more improvement is possible on microprogrammable machines with a large, fast buffer store. Where the machine architecture is appropriate, a firmware ladder implementation can provide an efficient APL system.

3.3 HARDWARE

The final ladder-based APL processor uses hardware specially designed for the task. In designing a processor with a specific application in mind, the designer can include features that other processors lack. For example, an additional computational element and the associated data paths can do all the computations for one iteration step simultaneously. An addition like this might have far-ranging effects on the rest of the processor; design options including this one will be discussed in detail in

Chapter 4. The rest of this chapter discusses the system design of a ladder-based APL processor that uses special hardware.

The ladder-based APL processor must not only execute ladder networks but also prepare them. Further, the processor must handle the core allocation for the APL arrays and input/output from any terminals and probably from a disk system as well. A computer with a conventional instruction set is a better tool for these tasks than a machine to execute ladders, with its very simple instructions. The instructions required to implement the ladder mechanism must reference only a small address space and need no addressing mode more complicated or powerful than direct addressing. (In direct addressing, a field in the instruction contains the address of the operand -- there are no index registers, indirect pointer chains, or any other addressing mechanism often found on a general-purpose digital computer.) If the ladder processor provided only those capabilities required to execute ladder code, it would be unable to support a total APL system. But, the capability to execute both ladder code and a more conventional instruction set can be provided in at least three ways.

First, only the conventional instruction set might be implemented, and the ladder code then implemented using these conventional instructions. This is simply the software approach to implementing an APL machine except that

a specially designed conventional machine replaces a commercially available one. To design a machine that already exists is pointless.

Second, the machine could have two operating modes -- one in which it executed ladder code and the other in which it executed the more conventional instruction set. This machine is more difficult to design than the first one because it requires the design of two machines -- one similar to the first and a simpler one for the ladder instructions. The performance of the machine in each mode is less than it might be because of the overhead that the necessary compatibility between the two modes forces on the design. There is little to recommend this technique.

Third, two machines could be used in the APL system, each machine corresponding to one of the modes of the machine in the second possibility. The conventional machine used in the third option can, of course, be a commercailly available computer; there is no reason why both it and the ladder machine should be specially designed. The machines that must be designed and built in the first two possibilities are more complicated than the ladder machine in the third. The first option would result in a less efficient implementation of the APL processor. The two machines in the third design are almost independent and can be run in parallel. The third approach is used here.

What tasks should be given to the specially designed ladder machine? What tasks should the conventional host machine do? For some of the tasks in processing APL programs, an immediate and clear-cut assignment is possible. The ladder machine executes ladder networks and the host machine translates APL programs into ladder networks. The host machine should also do other tasks like core allocation and I/O processing. For some tasks the assignment is not so straightforward -- the check to be sure that the ladder and splice code generated for a previous execution of the APL statement can be used again, the conformability check, and the transformations for the selection operators derived in Appendix B.

The ladder and splice code for a statement must be regenerated only if the ranks and types of the arrays referenced in the statement have changed. This check is performed each time a statement is executed. The rank and type of each array in a statement must be stored when code is generated for that statement. The ladder structure discussed above had no provision for storing data from statement to statement; so, to perform the check, the ladder structure must be greatly modified. It needs a mechanism for identifying statements and the memory for the old ranks and types. Since there is only one check per statement execution, the modifications are not cost-effective. The host machine must store the old ladder and splice code, so it can store the ranks and types along

with the code and do the check.

A similar argument applies to the conformability check. The code is executed only once per statement and, in general, it references the data-access parameters of all the ladders. In the structure outlined so far, each ladder could reference only the data-access parameters of its own array and not those of any other. This limitation allowed the ladder and splice code to use very small addresses, with the result that these instructions require only about twenty bits. Having the ladder processor check conformability would not significantly reduce the work done by the host machine because the host machine must load the conformability check instructions in the first place. Without the aid of special hardware, the time it takes for the host machine to load the instructions for the check into the ladder processor is comparable to the time it takes for the host machine to check, because the only loops in the conformability check code generally have very small repeat counts (the ranks of the arrays in the statement).

For the two architectures presented in Chapter 4 the type of loop required for the conformability check is very difficult to implement, so the machines proposed here would perform the conformability check using unfolded loops (straight-line code). Furthermore, since the ladder processor does not handle null arrays (the completion check occurs after the loop code has been executed), the shapes of

the arrays in the statement must be examined before the execution of the statement's ladder network can begin. So conformability checking is best left to the host machine.

Sometimes the conformability check cannot be done as the ladder machine is loaded because the statement must be partially executed before the shape of one of the intermediate results is known. In this case the statement is broken into statements that can be checked for conformability as the ladder and splice code for each smaller statement is loaded into the ladder machine.

The last of the tasks that might be assigned to the ladder machine is the execution of selection operators. Selection operators and conformability checking use similar computations. Except for the transpose operation, the maximum loop count is the rank of the array argument of the selection operator. Selection operations with constants or simple variables for the left argument can be performed when the ladder machine is loaded and, for the same reasons as for conformability checking, should be. When the left argument or subscript is an expression, the selection operation cannot be performed until the expresion has been evaluated. At the end of the evaluation, the ladder machine may perform the selection operation, or control may be returned briefly to the host machine so that it can perform the selection operation. The designs discussed in Chapter 4 use the host machine.

Where communication with the host machine is convenient and the operating system on the host machine can perform these small requests quickly, this approach is perfectly suitable. Chapter 4 explores adding the transformations in Appendix B to the ladder machine.

The two computers must communicate a lot. The ladder machine uses the data-access pointers to reference arrays stored in the main memory of the conventional machine. The conventional machine must load all the code and data into the ladder machine. The conventional machine must start and stop the ladder machine, determine its state, and read its memory. The ladder machine must inform the conventional machine that the execution of the ladder network is done. The ladder machine references the memory of the conventional machine only through the data-access pointer; a standard direct memory access (DMA) arrangement is suitable for these references.

Perhaps the most efficient way to let the conventional computer reference the memory of the ladder machine is to place the memory in the address space of the conventional machine. This makes it possible for the host machine to use its regular instruction set rather than its I/O instruction set to load the ladder-machine memory. In fact, if some of the contents of the ladder machine memory are regenerated each time they are loaded, then they can be generated in place instead of in the regular memory and transferred to

the ladder-machine memory. The host machine can reference the status information of the ladder machine in the same way it references the ladder-machine memory. The ladder machine should notify the host machine when the execution of the ladder network is done by means of an interrupt request.

A more detailed description of the communication between the host machine and the ladder processor requires more detailed knowledge of the system architecture of the host machine. Below we discuss two examples -- one using the PDP-11 as the host machine and the other using the PDP-10.

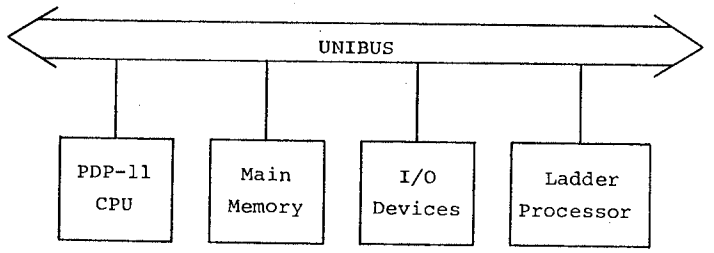
The connection between the ladder machine and the PDP-11 is very simple since all communication between components of a PDP-11 system is done over the UNIBUS -- see Figure 3-7. The cost for this simplicity is in system performance. When the ladder machine references a location in the memory of the PDP-11, the CPU cannot use the UNIBUS. Another limitation on system performance is that the data path width of the UNIBUS is only sixteen bits so that the transfer of a floating-point number requires two or four cycles depending on the precision. The PDP-11 is a suitable choice for the host processor in a moderate-performance APL system.

Other small computers besides the PDP-11 can be used in a moderate-performance APL system. For instance, the connections for an APL system using a Data General

Corporation Nova computer [DG 1972] could not be illustrated quite so simply as the system based on the PDP-11. But the actual complexity of the two systems (measured by the number of components to implement them) is about the same.

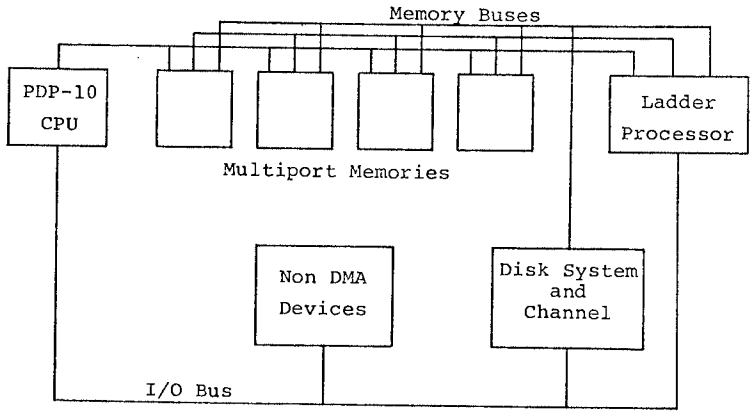
The connections in a system using a PDP-10 as the host processor are more complicated -- see Figure 3-8. The compensation for the added complexity of this configuration is a gain in system performance. The four memory buses can work simultaneously and, if the target memory units are distinct, the memory operations can be done in parallel. For this system, even if the ladder processor referenced memory at the maximum rate of a memory bus, the other memory buses would still be free and the memory would be available to the other components of the system an average of 75 per cent of the time. Furthermore, the data path width on the memory buses is 36 bits so that only extended-precision floating-point numbers require more than one memory cycle to access. The PDP-10 or a similar computer is an ideal choice for the host computer in a high-performance APL system.

Some quirks of the host machine might complicate the problem of connecting another processor into the system. The effect of cache memory, for instance, on the design of the ladder processor depends on the details of the cache. No serious problems should arise because an APL system with a host processor and a ladder processor is simpler than a multiprocessor configuration of processors like the host



Moderate-Performance APL System

Figure 3-7



High-Performance APL System

Figure 3-8

machine. The reason for this is that the ladder processor is always a slave processor, and only one processor controls the scheduling and allocating of system resources.

The designer will often have to limit the speed of the ladder processor so that its interference with the host machine (through competition for main memory cycles) is held to an acceptable level. In a stand-alone, single-user APL system, the ladder processor can tie up a large fraction of the memory cycles with little impact on overall system performance, but in a timesharing system the ladder processor should not degrade performance by more than a small amount, say ten or fifteen per cent. Several ladder machines can be placed on the same host machine; then the maximum allowable interference for one ladder machine on the host processor will have to be calculated assuming that all ladder processors are active at once. The most straightforward calculation indicates that the maximum tolerable interference of a single ladder processor is reduced by a factor equal to the number of ladder processors on the host machine.

This chapter discusses three implementation techniques for a ladder-based APL system.

The first uses only software. The performance of such a system is comparable to that of a standard software APL interpreter. It offers the advantage of using much less storage than the interpreter to implement some APL

operators. But a ladder-based system probably requires more code than an interpreter. So there are arguments in favor of using a software ladder-based APL processor rather than an interpreter, but the choice is not clear-cut.

The second uses firmware. If the architecture of a microprogrammable machine is suitable a microcoded ladder-based APL processor can give good performance. Even on a machine with an unsuitable architecture, special microprograms can improve the performance of a ladder-based APL system by a factor of two. But similar improvements are probably possible by adding special microprograms to a software interpreter.

The third uses a specially designed machine. Besides the special processor, the APL system includes a conventional general-purpose digital computer. The parallelism and expected performance of the special processor in executing ladder networks combine to make this technique an attractive one. Two designs of the special processor are explored in the next chapter.

CHAPTER 4

TWO LADDER PROCESSOR DESIGNS

Of the APL systems covered in the last chapter the one that performs best includes two processors, a conventional digital computer and a specially designed processor that executes ladder networks. This chapter examines the ladder processor in detail.

In the design of a processor, the designer first establishes the overall functional requirements (in this case, the capability to process ladder networks). Then he enumerates methods for implementing each part of the computer, guided by his intuition and his knowledge of the available components. He uses cost-benefit analysis to select the best of these methods for each part.

The whole procedure is less organized than this description suggests. Because the parts work together, many of the decisions are related; often the characteristics of other units in the complete system strongly affect the design. When the designer changes one unit, he should reconsider the design of the others. To illustrate both the design procedure and the flexibility of the two-processor approach, this chapter and the associated appendices spell out two designs for a ladder processor. One stresses high performance; the other stresses simplicity. Options for these processors illustrate cost-benefit analysis.

4.1 HIGH-PERFORMANCE DESIGN

One of the first steps in designing a computer is to determine the representation of the data to be processed. Decisions about the data representation are, on the whole, independent of decisions about the control organization of the machine. With the data representation fixed, the widths of data paths are known and thus their costs can be evaluated. The ladder processor uses two general classifications of data, ladder data and array data. The ladder data, which are required to operate the ladder mechanism, are: the base address of the array, BETA; the data-access pointer, PI; the rank; and the subscript, shape, step, and separation for each coordinate. The array data, stored in the temporaries, are used in the calculation explicitely specified by the APL statement.

All ladder data are integers so that the only problem in representing these data is determining their precision. There is only one value each for the rank, the type, the program counter, the base address, and the data-access pointer for each ladder. There are several values for each of the remaining ladder-data items: the subscript, the separation of elements along the coordinate, the shape vector component, and the DELTA value.

As the study by Saal and Weiss [1975] showed, arrays with a rank greater than two are uncommon. In their sample, references to arrays of rank three made up less than one

percent of the array references and they observed no reference to an array of rank four or greater. Because they counted the number of subscript positions in subscripted array references to determine the distribution of ranks, these figures may not precisely represent the total distribution of ranks of arrays. Nevertheless, they strongly suggest that arrays of rank greater than three are very rare. We can conclude that the capability to represent directly arrays with a rank up to six is adequate to process almost all APL programs. (Six because four data-access parameters for each of six coordinates plus up to eight parameters not duplicated for each coordinate can be addressed with five bits.) With the addition of one bit in the address a rank of fourteen can be handled. The cost of more than doubling the number of ranks that can be represented is much more than the cost of adding this bit to each address of a data-access parameter. The storage for all data items that have a separate value for each coordinate of each array is doubled. This storage is by far the largest component of the storage for ladder data and array data combined. Therefore, doubling the range of values of the rank of an array almost doubles the data storage required by the ladder processor. The expected gain is almost nothing since the vast majority of arrays have a rank of less than four and surely almost all have a rank of less than seven. The expected number of arrays with a rank larger than six and and less than fifteen is very close to

zero. It is impossible to justify extending the range of ranks handled directly by the ladder processor past six because the costs far outweigh the expected benefits.

APL has only two basic data types, character and numeric. The numeric data type can be further divided into a logical class, where values are limited to zero and one, and all other numbers. For computational purposes, numbers are represented in one of two ways, fixed or floating point. The type information is used to control the generation of an address of a data element in the main memory of the host machine and to select which computation unit to use in an arithmetic operation.

Each ladder has its own program counter, used to cycle through both the ladder and the splice code. If a specialized instruction set is used, then for each coordinate there is one instruction to initialize the subscript and one instruction to step the subscript and data-access pointer and check for completion in that coordinate. There must be two additional instructions -- one to initialize the data-access pointer and the other to return control to the first instruction of the ladder when the outermost loop has completed. Thus, the number of ladder instructions used for an array is two times the rank of the array plus two. The number of splice instructions necessary to implement an APL statement varies with the operators in the statement. For the simple scalar

operators, one or two instructions are generally sufficient, and for the more complicated operators, two or three times this number suffice. A total of 32 instructions for the splice and the ladder code will be enough for almost all cases.

Saal and Weiss found that the average number of nodes in the parse trees of the APL statements in their sample was nine and that the maximum was 35. In the worst case about half of the nodes can represent an array reference. The capability to process a network of 32 ladders will enable the ladder processor to execute an APL statement about twice as large as the largest Saal and Weiss found. The extension of the maximum number of ladders allowed in the network from 32 to 64 will enable very few additional APL statements to be executed whole rather than broken up. This extension will allow some APL statements to be elided and executed as a single statement. Of course, except in special circumstances, it will not be possible to elide two statements where the output of one statement is used as an input to the next. When statements can be elided, there are fewer places in the APL program where the ladder processor and the host machine must interact. The work directly associated with these interactions (conformability checks, storage allocation, loading of the ladder processor, and so on) is not reduced, but the overhead in initiating and scheduling these interactions is. On a timesharing system this overhead can be expensive. Increasing the capability

to execute a ladder network from 32 to 64 will not noticeably affect the number of APL statements that the ladder processor can handle. But the reduction in overhead gained by eliding statements merits the expanded capability.

The memory allocated for code used by the ladder network is 64 instructions each for 64 ladders for a total of 4K storage locations. A program counter of twelve bits is required to reference 4K of code. It is possible to use a program counter of six bits and append to it the six-bit current ladder number to form the twelve-bit instruction address. This instruction addressing technique partitions the instruction memory into 64 fixed segments of 64 locations each. Each ladder can reference instructions only within its segment. In some APL statements the splice code for several operators is inserted into the same ladder. This results in a small amount of code for several ladders while one ladder has a great deal of code. If the instruction memory were divided into fixed partitions (as with a six-bit program counter), then the code for one of the ladders could overflow the code partition for that ladder, and the ladder network would have to be split into separate networks to be executed. For this reason the ladder processor uses a twelve-bit program counter. The flexibility provided by a twelve-bit program counter allows this uneven distribution of code to be handled naturally.

The remaining two quantities are the data-access pointer and the base address of the array. These quantities can be treated together since they both specify the same thing -- the address of an element of an array. Clearly they both must be able to reference any location in the memory of the host machine, and any addressing capability beyond that is useless (assuming that the ladder processor is for use with only one machine and no expansion of the addressing capability of the host machine is contemplated). The number of bits required by PI and BETA is not necessarily the same as the number of bits in a physical address of the host machine. One of the data types of the ladder processor is logical, and only one bit is required to store a logical data item. In most computers many logical data elements can be stored in the smallest addressable unit. (There are eight bits in one byte.) The address-sequencing mechanism is the same for all data types. Two adjacent logical data items might reside in the same byte (or word) and thus have the same physical address while two adjacent non-logical data items might reside in adjacent bytes (or words) and thus have different addresses. The interface between the ladder processor and the memory bus of the host machine translates a data item's address generated by a ladder to its physical address in the memory of the host machine. The data type of the element referenced is used in the address translation. The number of data elements stored in the smallest addressable unit of the host

machine is limited to a power of two so that the address translation is as simple and as fast as possible. The number of bits required for PI and BETA equals the number of bits in a physical address of the host machine plus the floor of the log base two of the number of bits in the smallest addressable unit. For a PDP-10 system based on a KA-10 processor:

$$N = 18 + \lfloor \log_2 36 = 23$$

For an IBM 360 or 370 with a 24-bit physical address:

$$N = 24 + \lfloor \log_2 8 = 27$$

There need be no limit on the number of elements along any dimension of an array short of the limit imposed by the availability of storage. This does not necessarily mean that the same precision must be used in the subscript and shape vectors as in the data-access pointer and the base address. In some multiprogramming systems no user is given access to all physical address space. For instance, on the PDP-11 a virtual address is sixteen bits, so a user can reference an address space of 64K bytes. But the physical address space of the PDP-11 is 256K bytes. PI and BETA must be large enough to reference any bit in 256K bytes (21 bits are required) but the maximum length of any vector a user can create is 64K bytes or 512K bits (a nineteen-bit subscript and shape vector element are sufficient). The number of bits required to represent PI and BETA is

certainly an upper bound on the number of bits required by subscripts and shape vector components. In systems where the user is provided full access to the physical address space (after passing through a memory management or memory protection unit) the same number of bits are required to represent the two different types of data. Even in systems where the user does not have full access to physical address space, appropriate extensions to the operating system can provide this capability. Therefore, where such modifications are considered possible, the ladder processor should be designed as if the user could reference all physical address space.

Some transformations of the type described in Appendix B require the same precision for DELTA and G components as for subscripts. For instance, if a logical vector fills all the data storage available to a user, a subscript operation that selects only the first and the last element (an arithmetic progression with only two elements) will set DELTA1 and G1 equal to one less than that maximum length.

Because the temporaries store data manipulated by both the ladder processor and the host machine, the data representation used in the temporaries is dictated by the host machine. For a ladder processor connected to a PDP-10, a logical choice for the data representations used in the temporaries might be 36-bit integers and floating point numbers, nine-bit characters with four stored in each 36-bit

word, and single-bit logical data with 32 stored per word. The precision used in the temporaries is generally greater than that used in the ladder data so that the widths of the general computational data paths are determined by the precision of the temporaries.

Before he can start on the internal structure of a computer, the designer must choose an instruction set. In the ladder processor, there are two instruction sets, the ladder code and the splice code.

The address-sequencing mechanism of the ladder processor has only four indivisible actions -- initializing the data-access pointer, initializing a subscript, branching to the beginning of the cycle, and the iteration step. The implementation of this mechanism in Chapter 3 showed that the first three actions are very simple. Each used a single PDP-11 instruction. The iteration step required five.

Among ladders for arrays of equal rank the only variation comes in the splice code. The structure of ladders of the same size can be made identical by placing the splice code elsewhere in memory and using subroutine jump and return instructions to go between ladder and splice code. With subroutine jumps in all the splice positions, the relative position of the instructions in the address-sequencing mechanism is fixed.

The code below is the ladder code for an array of rank three. An ITERATION STEP N instruction updates the Nth subscript and the data-access pointer and branches to LOOPN if the Nth subscript is less than the Nth component of the shape vector. Otherwise it passes control to the next statement. The LOOP instruction is an unconditional branch to LOOP0.

```
LOOP0:  SPLICE JUMP 1
        BETA PI
        I1←1
LOOP1:  SPLICE JUMP 2
        I2←1
LOOP2:  SPLICE JUMP 3
        I3←1
LOOP3:  SPLICE JUMP 4
        ITERATION STEP 3
        SPLICE JUMP 5
        ITERATION STEP 2
        SPLICE JUMP 6
        ITERATION STEP 1
        SPLICE JUMP 7
        LOOP
```

The branch offset for the LOOP instruction is four times the rank plus three, and the branch offset for the iteration step of the kth coordinate is (rank-k)×4+2. Of the five instructions used in the code above, two need not specify any parameters and two must specify coordinate number. The fifth instruction, the splice jump, must specify the location of the splice code. This can be done by a table with the location of the splice code for each splice-code jump. Using this technique, the designer can encode the ladder code for an array of rank six in five bits.

It is possible to store the branch offsets rather than the splice-code locations in a table. This option has several benefits. The splice code can be placed inline and the splice-jump instruction can be eliminated. The size of the table required is cut in half, and the remaining four ladder-code instructions can be encoded in four bits for arrays of rank seven.

The instruction size of the splice code, however, will definitely be greater than four bits, so the embedding of splice code in the ladder code carries with it a significant disadvantage. Processing two instructions of different length requires a substantial amount of additional logic. Instruction decoding requires two separate sets of logic. There must be a way to select which of the two instruction decoders should be used for each instruction. This problem is solvable but the solution results in further complications in the design of the ladder processor. The benefit provided by allowing two instruction formats is that memory is saved because of the short ladder instructions. (64 arrays of rank six require almost 1000 instructions in their address-sequencing mechanisms.) The savings are more significant for the host machine because the ladder machine stores only one statement at a time. But the host machine can regenerate the ladder code very easily and so need not store it. Therefore, to simplify the instruction decoding logic, the splice code and the ladder code will be the same length.

The splice code is used to implement the calculations in an APL statement. It is desirable to make this code as compact as possible consistent with the goal of high execution speed. With this policy sequences of simple instructions that implement an APL operator should be collapsed into a single instruction. In some circumstances, the expected savings derived from these compound instructions are not sufficient to justify the costs in providing the instruction. The costs are not necessarily directly related to the complexity of the computation required by the instruction. For example, in a microprogrammed processor, instructions to perform the trigonometric functions might be simple to provide. These instructions easily fit in the format of a standard single-operand instruction. The difficulty associated with them is in writing and providing storage for the microprogram to implement them. Writing the microprograms to implement these instructions is not tremendously more difficult than writing the software to accomplish the same calculations. It is possible that the storage required for the microprograms to implement these instructions will, for technical or economic reasons, force the use of denser but slower memory components. In this way the provision of complex instructions in the instruction set of a microprogrammed processor can reduce the overall performance of the processor. Here the design provides enough high-performance microprogram storage so that all

microprograms fit. On the other hand, the expected gains are small (both in time and in space) because these complex instructions are infrequently used [Bingham 1975].

Instructions are costly to implement if they do not fit into the formats already provided. Instructions of this type often require additional instruction decoding logic and extensions of data and control paths. This additional hardware will almost certainly result in the slower processing of all the instructions. Examples of this type of instruction are given below.

Only four instructions are needed to implement the ladder mechanism, and to simplify decoding they should be similar in format to the splice instructions. The splice code should closely match the APL operators to minimize the amount of splice code generated. It is easy to design an instruction format for the scalar operators. Rather than add formats, operators that cannot be implemented with one instruction in this format should be implemented with several.

An instruction must specify the operands as well as the operation to be performed. There are four major choices of methods to specify the operands -- code using zero, one, two, and three addresses.

In a machine that uses zero-address code, the operands are specified implicitly. The operands are the top two elements on the operand stack. Thus an ADD instruction would replace the top two elements of the operand stack with their sum. This allows instructions to be short because no addresses must be specified except in PUSH and POP instructions. Zero-address code has disadvantages. Consider the following APL statement:

A←B+C+D+E

where + can be any scalar operator. In a zero-address machine each operator is implemented by a two-instruction sequence: PUSH @PI; ADD. These instructions require at least two memory cycle times to execute, but other types of instructions can perform the same computation in one. Since the primary goal of the high-performance ladder processor is to execute the code corresponding to an APL statement in minimum time, zero-address code does not appear to be the appropriate choice for the ladder processor instructions.

A single-address instruction specifies one operand; the remaining operand is specified implicitly and is usually located in an accumulator. The accumulator is loaded with the result of the operation. Single-address code is well suited to the example discussed in connection with zero-address code. But consider the following example:

A←(B×C)+D×E.

The single-address code to implement this statement requires seven instructions while the double-address code described below requires six instructions. The time required for the two types of instructions is appproximately the same on any computer with an architecture similar to the ladder processor. So the single-address splice code for the statement above requires between fifteen and twenty per cent more time to execute than the double-address code.

A double-operand instruction specifies both of the operands; the contents of one of the operands (called the destination operand) are replaced by the results of the instruction. The double-address code for the first example APL statement requires the same time as the single-address code, and for the second example the double-address code requires less time.

In triple-address code the addresses of the two operands as well as the address where the result is to be stored are specified by the instruction. The two input operands are not changed by the execution of a triple-address statement. Although it is sometimes efficient to save one of the operands, it is rare that both operands need be saved. Therefore, the double-address code will be used in the ladder processor. To ensure that either of the two operands in a double-address instruction can be retained, there are two instructions for each of the common non-commutative operators. SUB A,B is equivalent to B←B-A

and RSUB A,B is equivalent to B←A-B.

All the instructions of the ladder processor are listed in Appendix C. There is an instruction for every primitive APL function with scalar inputs and a scalar output. The remaining APL operators are implemented with a sequence of splice code (sometimes spread over several splices), are implemented by the host machine (grade up, grade down, and I beam functions), or are selection operators and are implemented as described in Appendix B.

Some APL operators have no corresponding instruction in Appendix C. For several of these operators the following discussion tries to show why special instructions were not included in the instruction set of the ladder processor for them. In all cases the decision is based on a comparison of the cost to include a special instruction with the benefits that instruction would provide.

The first instruction to be examined is decode, or base value. Decode is a dyadic function and produces a numeric scalar as output. The result is the number whose representation is given by the right argument. The left argument specifies the radix system of the representation. The accumulation of the result is done in a loop and the repeat count for the loop is the length of the argument vectors. In one pass through the loop the current value of the result is multiplied by the next component of the left argument (the radix vector) and then the corresponding

element of the right vector is added to the result. Three quantities are referenced within the loop. Since double-address instructions can reference only two quantities, the loop must contain more than one instruction. The three-address code discussed above is also not suitable for this computation. In three-address code the result of the application of the operation specified by the instruction to the two input operands is placed in the location specified by the third address. In the computation in the decode loop, first one operation is performed between one of the input operands and the result operand, then another operation is performed between the other input operand and the result operand. One three-address instruction could select the three operands, but the architecture of a standard three-address machine must be considerably extended to allow the type of instruction required by the decode loop. Since this computation can be performed by two standard double-address instructions, the decode operator does not provide enough justification to shift to an extended three-address instruction set. It is not surprising that the decode operator is not directly compatible with a double operand instruction set since it is really a polyadic function.

The encode (representation) operator is the inverse of the decode operator. The left argument is the radix vector just as before but the right argument is a numeric scalar. The result is the representation of the right argument in

the radix system specified by the left argument. For each component of the result, the calculation given by the following ALGOL statement is performed.

```
IF left component = 0 THEN
BEGIN
  result component := right arg;
  right arg := 0;
END
ELSE
BEGIN
  result component := right arg REM left arg;
  right arg := right arg DIV left component;
END
```

Just as with the decode operator, the solution step uses three variables. Two of the variables are outputs. The structure of the decode operator fits the standard three-address model poorly; this is even worse. Unfortunately, the implementation of encode with the instructions in Appendix C is more complicated than the implementation of decode. Benefits result from expanding the organization of the ladder processor to implement one solution step of the encode operator with a single instruction. These benefits are greater than for the decode operator, but the modifications required are much more extensive. Just as for decode a special instruction is not cost effective.

The monadic iota function, called index generator, produces a vector of integers whose length is given by the scalar argument. To produce one component of the output from the previous one, the splice code simply adds one. But

the temporary used to hold the output component might be modified by computation following the monadic iota. The modification of the output temporary of a ladder is generally of no concern since the ladder will reload the temporary when it produces the next component of its output. The old contents of the temporary are not used in any way to produce the new output. The reuse of temporaries simplifies splice code and reduces the number of temporary storage locations required by the splice code of an APL statement. The output of the monadic iota must be buffered to shield the running value of the index from the output temporary. The use of three-address code throughout the ladder network would eliminate the need for the shielding, but it would result in a substantial increase in the number of temporaries used. There are two approaches to implementing monadic iota.

If the monadic iota function is implemented by creating a dummy ladder to step through the elements of the result vector, then the subscript maintained by this ladder can be used as the running index. The shielding is provided by an instruction that loads a temporary with the subscript. This step is required even if there is no need for shielding, since the splice code in one ladder cannot reference the subscripts of another ladder. The other approach calls for the addition of a double-operand instruction. The instruction adds one to the input operand and stores the result in the output operand and the input operand. Because

the result must be stored in both operands, such an instruction requires some addition to the architecture in Appendix C. This special instruction would remove the necessity of tying the output of the monadic iota function to the subscript of a ladder and thus would allow the ladder network generator more flexibility in its optimization strategy. (Consider the APL expression ⍳N+A/M.) The elimination of one instruction from the code for monadic iota, and then only in special cases, does not produce enough of a gain to merit the changes required.

A similar analysis of dyadic iota and dyadic epsilon indicates that these two functions are best implemented with several splice-code instructions contained in Appendix C. No special instruction is appropriate for either of these functions. The number of parameters that must be specified in such an instruction is so large that none of the existing instruction formats is suitable. The expected benefit of a special instruction for either of these two operators does not justify the expense of adding a new instruction format and all the additions to the data paths required to support the new format.

There is only one set of arithmetic instructions, not one for integers and another for floating point numbers. In order to simplify the task of generating ladder networks, the ladder processor maintains type information for all the temporaries and automatically uses the correct arithmetic

unit for any operation on them. Type conversions are also performed automatically, but this requires that the ladder processor have other capabilities. For instance, in the statement A←B×C where B and C are integer arrays, the components of the product would be stored as integers if there were no overflow. An overflow might, however, occur after a number of the result components have been stored. Since the array stored in the host machine must be homogeneous, the ladder processor must be capable of stopping the processing of the ladder network and informing the host machine of the overflow. The computation can be restarted, but a conversion of the results to floating point is forced.

The organization of the ladder processor allows all computations or conversions that might be first steps of instructions to begin in parallel. In this way the time required to decode the correct sequence of computations and conversions is not wasted but is overlapped with the calculation time. Of course, the decoding must be complete before any result is stored.

The speed of the logic circuitry used in computer systems today, from stand-alone minicomputers costing a few thousand dollars to large multi-processor configurations costing several million dollars, varies by about a factor of ten [Bell 1971, TI 1973, TI 1974, AMD 1, Motorola 1974]. Generally, the parallelism of the design of a large system

accounts for considerably more of the performance gain over a smaller system than does the speed of the logic of the two systems. Parallelism plays a very important role in the high-performance ladder machine. As the block diagrams in Appendix C show, the iteration-step instruction uses four separate adders. These adders and the associated memories operate in parallel so that the iteration step can be performed in one memory access time plus two add times. Because there are two copies of the temporary storage locations, it is possible to fetch both operands of a double-operand instruction at the same time. During the write phase of the instruction both memories are updated so that they always contain the same data. Furthermore, the instructions are stored in a separate memory so that the next instruction can be fetched during the execution of the current instruction (except for a branch). The parallelism and speed requirements of the design shown in Appendix C are expensive. Memory components sixteen times as dense as the ones used for the ladder data could have been used if it were not necessary to fetch several data items simultaneously during the execution of the iteration step instruction.

The cost would have been much higher if some restrictions had not been placed on the capabilities of the ladder processor. For instance, two different subscripts cannot be the operands of a double-operand instruction -- an ADD I1,I2 instruction executes as if it were an ADD I2,I2.

The reason is that a double-operand instruction fetches both operands in parallel but only one coordinate number may be used in forming the address to the ladder-data memory. There are two ways to eliminate this restriction. The first approach is to provide two copies of the ladder data just as with the temporaries. The cost of this method is entirely out of proportion with the expected benefits. The other approach is to decode the fact that two conflicting ladder items have been referenced in one instruction. Instead of going to the normal microcode for the instruction, control passes to a special microprogram that splits the execution of the instruction into two parts. In each of the two parts only one data item from the ladder-data memory is referenced. The cost of this approach is not prohibitive. But such conflicts arise so infrequently that it is probably best for the designer to make no provision for them in the ladder processor. The splice-code generator can produce two separate instructions in the problem cases.

The high-performance ladder processor proposed here is not fully microprogrammed. The iteration step, coroutine jump, branch, and halt instructions are all hardwired, as is the instruction fetch cycle. This split was included in the design because it offered a speed advantage over a fully microprogrammed processor. And, since all the hardware was provided to do the calculations in parallel, hardwired control was not difficult to implement.

The fields contained in each microinstruction along with several sample microprograms to illustrate their use are also given in Appendix C. The sample microprograms indicate that most of the macroinstructions can be implemented with microprograms one or two microinstructions long. Two longer microprograms illustrate the use of some of the microprogram control fields that the microprograms of the simple instructions do not use. The code segments given in Figures 4-1 and 4-2 are symbolic representations of these microprograms.

In the code segment in Figure 4-1, each line represents one microinstruction. The conditional branches test conditions set by the previous microinstruction. The code in the same instruction as a branch is executed whether the branch condition is satisfied or not. A STOP in a microinstruction normally causes a return to the hardwired macroinstruction sequencer. If a STOP and a branch are in the same microinstruction, a return to the instruction sequencer takes place only if the branch is unsuccessful.


```
        TEMP1←FLOOR(A)
        TEMP1-A                 BEQ ZERO
        TEMP2←TEMP1             BNE GAMA
LOOP:   2-TEMP1
        TEMP1←TEMP1-1           BGE DONE
        TEMP2←TEMP2 TEMP1       BR LOOP
DONE:   A←TEMP2                                     STOP
ZERO:                           BNE GAMA
        A←1                                         STOP
GAMA:
```

Factorial (!A,Integer part only)

Figure 4-1

In Figure 4-2, the two instructions immediately before the instruction labeled CONT appear as though they might be collapsed into a single microinstruction. These two instructions transfer DELTA1 to G1. This cannot be done in one microinstruction (unless the microinstruction is executing a MOV DELTA1,G1 macroinstruction) because only one field in the microinstruction can specify a ladder datum. This one field can't reference two different ladder data. Of course, these two instructions could be replaced by the following microinstruction:

```
T2←DELTA1               BMI RANK1
```

This illustrates a limitation the architecture of the ladder processor imposes on the computations that can be done in one microinstruction.

A design first used by Wilkes [1953] suggests that the ladder processor would perform better if it were divided into two processors that run in parallel. Unfortunately, this approach has hidden difficulties. The splice code as

```
        T1←RANK-1
        T1←T1-1          BMI SCALAR     (AN ERROR)
        T2←DELTA1        BPL CONT
        G1←T2                           RETURN
CONT:   T1←T1-1
        T2←DELTA2        BMI RANK2
        T1←T1-1
        T2←DELTA3        BMI RANK3
        T1←T1-1
        T2←DELTA4        BMI RANK4
        T1←T1-1
        T2←DELTA5        BMI RANK5
        T2←DELTA6
        G6←T2
        T2←T2×RHO6
        T2←T2+DELTA5
        T2←T2-DELTA6
RANK5:  G5←T2
        T2←T2×RHO5
        T2←T2+DELTA4
        T2←T2-DELTA5
RANK4:  G5←T2
        T2←T2×RHO4
        T2←T2+DELTA3
        T2←T2-DELTA4
RANK3:  G3←T2
        T2←T2×RHO3
        T2←T2+DELTA2
        T2←T2-DELTA3
RANK2:  G2←T2
        T2←T2×RHO2
        T2←T2+DELAT1
        T2←T2-DELTA2
RANK1:  G1←T2                           RETURN
```

Microprogram to Compute the Components of the G Vector from the DELTA and RHO Vectors

Figure 4-2

well as the ladder code read and write the ladder data. Suppose that we would like to execute the ladder-code iteration step concurrently with the splice code that normally would be executed just before it. Since this pattern of splice code followed by an iteration step instruction is the structure of the innermost loop of every ladder, it is with just such an overlap that we would expect to save the most time. The iteration-step instruction computes the new values of the subscript and the data-access pointer and it determines whether the splice code is executed again. The ladder processor may not update the values of the ladder data because a splice code reference to this data must read the old values. The innermost splice code will very likely contain a coroutine jump. The new ladder data still may not be written into memory. At the coroutine jump the ladder processor may save the newly computed ladder data, but memory has to be added for this purpose. Or it may throw them away and recompute them when control returns to this splice. In the simple ladder networks, where overlapped execution of ladder and splice code should result in the most improvement, it is highly likely that the instruction to be executed upon return is the iteration-step instruction. Since recomputing the data probably takes about as long as executing the iteration-step instruction in the non-parallel processor, little time has been saved. If we elect to save the new ladder data for a splice when we encounter a coroutine jump and then restore

this data when control returns to the splice, we must provide storage for the new data. The amount of storage is large because there must be room to store a new set of ladder data for each possible ladder. Thus it appears that the time saved, if any, does not justify the additional complexity of the machine.

Chapter 3 argued that the host machine should perform the transformations associated with the selection operators described in Appendix B. In some situations it would be advantageous if the ladder processor could perform the transformations as well. The conditions in which this can be done are discussed below.

The rank of the result of the transpose operator is determined by the value of the left argument. The ladder network generator must know the rank of an array to create a ladder for it. The processing of the left argument necessary to determine the rank of the result is similar to the computation required for the transformation for transpose. If the value of the left argument is known at the time the ladder network is generated, the host processor should perform the transpose selection operation. If the left argument is an expression, its value will not be known until execution time. The ladder processor can perform the transformation at execution time, but with several disadvantages and restrictions.

These calculations involve multiple passes over the left argument. (The number of passes is equal to one plus the rank of the right argument.) The left argument is an expression and the calculations necessary to evaluate it are done several times. For simple expressions this might be an acceptable burden but as the expression becomes more and more complicated this overhead can no longer be tolerated.

For the transformation to be performed at all, the right argument must be an array reference rather than an expression. The beating and drag-along techniques described by Abrams [1970] can be used by the ladder network generator to map statements into this form.

If the selection-operator transformation is to be done by the ladder processor, the ladder network generator must have generated a ladder for the right-argument array, and the rank of this ladder must be equal to the rank of the array. After the new rank has been calculated, the components of the shape vector for the eliminated coordinates must be set to one. If all the other ladder data are updated as described in Appendix B, then the resulting ladder will properly sequence through the elements of the array. But, since the rank of the result of the transpose is not known until part way through the execution of the statement, the ladder processor generally cannot perform the transpose transformation. The host machine should handle the transpose.

The case of the subscript selection operator is similar to that of transpose. There is no completely acceptable solution in which the ladder processor can do all the calculations for the transformation, so having the host machine perform the transformation seems best.

On the other hand, take, drop, and reverse can all be handled in a straightforward manner within the ladder mechanism. Just as for all the other selection operators the right argument of the reverse operator must be a simple array. The ladder for this array has a coroutine jump in the first splice. The ladder network that is the target of this coroutine jump yields the coordinate on which to perform the reverse operation. Splice code immediately following the coroutine jump negates the G value for the reverse coordinate and then computes the DELTA vector. At this point the transformation is complete and the remainder of the ladder sequences through the elements of the reversed array in ravel order.

The take and drop transformations are implemented in a similar fashion. Code that does a coroutine jump to get each element of the left argument is placed in the first splice. When an element is delivered, the transformation it produces is computed and applied. When all the elements of the left argument have been delivered, the DELTA vector is computed and the transformation is complete. The number of elements in the left argument is equal to the rank of the

right argument. The coroutine jump and transformation code can be placed in a loop with a repeat count equal to the rank of the right array, or, since the amount of code involved is quite small, all the code can be placed inline. The instructions listed in Appendix C are sufficient to implement reverse, take, and drop, and to compute the DELTA vector from the RHO and G vectors (and vice versa).

The ladder structure was designed to allow efficient sequential accessing of array elements. Some APL operators can be implemented with very little work within this structure. There are other APL operators that do not mesh nicely with the ladder structure, but they do not greatly impair the performance of a ladder-based APL system.

Chapter 5 gives an estimate of both the cost and performance of the high-performance ladder processor.

4.2 THE MODERATE-PERFORMANCE LADDER PROCESSOR

The primary design goal of the moderate-performance ladder processor is low cost. Sharing data paths and computational elements produces the most significant savings. Most of the parallelism of the high-performance ladder processor is eliminated. Chapter 5 examines both the reduction in cost and the increase of execution time that the elimination of most of the parallelism entails.

The discussion of the precision required for the various types of data applies here. The one parameter that changes is the number of ladders that can be processed in one network. The number is reduced from 64 to 32. As the previous discussion argued, this capacity is sufficient for the vast majority of APL statements. This reduction makes feasible a 4K limit on all the storage used by the ladder processor (all types of code and all types of data). The organization of the densest available random access memories is 4K by 1. In the next year or so 16K memories will become available, but the current 4K memories are considerably faster than the initial versions of the denser memory will be. The high-performance ladder processor has a memory of between 6K and 7K words. Thus the number of words per ladder is greater on the more modest machine.

In the previous design many of the data paths, computational elements, and memories served only one function. In such a situation the widths of each of these system components can be determined by the function it is to perform. In the more modest processor, there is no such specialization. There is only one arithmetic logic unit and one memory, and each data path has many functions. The most demanding use of these system components determines their size. Just as before, the temporaries require the widest data paths.

The most likely choices for the host machine in a moderate-performance APL system have word lengths of sixteen or 32 bits. Since a width of sixteen bits is certainly not sufficient to provide the required precision for the array data, the logical choice for the precision of the ladder processor is 32 bits. This precision is more than adequate for all the ladder data discussed previously. The precision provided by 32-bit floating-point numbers is insufficient for some numerical applications. For this reason the ladder processor must be capable of processing 64-bit floating-point numbers in addition to the normal 32-bit floating-point numbers.

The arguments for using a common instruction set for ladder and splice code are even stronger for the moderate-performance ladder processor than for the high-performance version. There are two readily apparent choices for instruction length, sixteen or 32 bits. It is possible to fit an acceptable instruction set in sixteen bits. The possible instruction types are the same as before, and the two most promising are single-address and double-address codes. A compromise between these two is also possible -- single-address code with multiple accumulators. The probable partition of a sixteen-bit double-address instruction into fields is:

4-bit op code field
6-bit source-address field
6-bit destination-address field.

The six-bit address fields make it possible to reference 32 ladder-data items (the same number as in the high-performance ladder processor) and 32 temporaries. The four-bit op code field allows fifteen double-operand instructions with the sixteenth code reserved to indicate instruction types other than double-operand. Both the number of double-operand instructions and the number of temporaries are so small as to be a disadvantage. Single-address code suffers from the disadvantages discussed previously, but it does allow the extension of the address and the op code fields. The addition of multiple accumulators can reduce the number of instructions required. While any of these three approaches is feasible, the instruction length used in this design is 32 bits.

There are two reasons for this choice. The first is that an instruction length equal to the word size allows simpler instruction sequencing. An instruction address specified by the program counter is the same as any other address and no choice between fetching a new instruction word or using the other half of the previous instruction word is required. The second reason is that the larger fields available for each purpose allow the op codes and addresses to be specified in a more easily decoded format. For example, the high-performance ladder processor appended the current ladder number to the five-bit ladder-data addresses used in that machine to form the physical memory address. In this machine, where simplicity is the primary

goal, the computational element would perform such a task serially. The 32-bit instruction allows enough room for each of the double addresses so that the physical memory address can be given directly. Furthermore, op codes can be given in partially decoded form.

For the moderate-performance ladder processor to do automatic type checking and conversion, the microprogram for each arithmetic operation would have to include microinstructions to check the types of the operands to decide which microroutine should perform the operation and whether any type conversions are necessary beforehand. To eliminate the need for these additional microinstructions, this processor has type conversion instructions and separate arithmetic instructions for integer and floating-point representations. Of course, this increases the difficulty of generating the ladder network, but the microinstructions necessary to do the checking would result in an increase by a factor of two in the execution time of the integer operations. The instruction set of the moderate-performance APL machine is listed in Appendix D. In addition, Appendix D contains a block diagram of the processor and some sample microprograms.

Several design changes could produce a faster processor. Adding control logic makes it possible to eliminate some microinstructions. For instance, additional timing logic and some changes in the method of handling

microprogram branches make it possible to merge the two microinstructions in the instruction fetch and initial decode sequence. Also, the decision whether to store the output of a single-operand or a double-operand instruction in local memory or host memory can be controlled directly by the destination field rather than by the microprogram branch, as in the examples. This technique would save one microinstruction per instruction execution. Each change would result in a saving of ten to fifteen per cent of the execution time of every instruction at the expense of the extra logic. The overriding goal in this processor is simplicity, so neither enhancement is used.

CHAPTER 5

EVALUATION OF TWO DESIGNS OF THE LADDER PROCESSOR

5.1 COST OF THE TWO DESIGNS

First we consider the cost of the high-performance ladder processor. Figures C-2 and C-3 give the logical organization of the processor. Once the logical design of a machine has been fixed, the physical design, based on available components, is a straightforward but sometimes intricate process. In all currently popular solid-state logic families, components that perform high-level functions are available. These functions include random-access memories, registers, multiplexers, and arithmetic logic units.

The first step in the physical design is the selection of a logic family. Of the numerous solid-state logic types available today, two are suitable for the construction of high-performance digital equipment. These two types are Schottky clamped transistor-transistor logic (commonly know as Schottky TTL) and the 10,000 family of the emitter-coupled logic families (commonly known as MECL 10,000). The propagation delay of a single Schottky TTL gate is about three nanoseconds and the delay through a MECL 10,000 gate is about two nanoseconds. Two features of ECL logic enhance this speed advantage. The basic building block of the MECL 10,000 line is the NOR gate, but, because of the circuit used to implement it, the complement of the

gate output is also available. This eliminates the need for inverter units and the propagation delays they introduce. The second advantage of the MECL 10,000 family is that the outputs are driven by emitter-follower transistors with open emitters. This makes it possible to connect the outputs of two logic components; the output is the OR of the two original outputs. (This technique is called wired-OR.) These two features make it easy to drive long cables (longer than about ten inches) with very little electrical noise [Blood 1972].

There are some disadvantages in the use of MECL 10,000 circuits. A termination network must be used for any line longer than a few inches. These termination circuits add to the component count, increase power dissipation, and lower the circuit density. Another disadvantage of MECL 10,000 with respect to Schottky TTL is that there are fewer logic functions available in MECL 10,000 than in Schottky TTL. This difference in component availability is most apparent in the multiplier units, in the floating-point computation unit, and in the floating-to-integer and integer-to-floating conversion units. Both logic families contain components specifically designed to aid the implementation of multiplication algorithms. One choice of TTL components allows about four times the density of ECL components but takes almost four times as long to compute the result. Different implementation techniques yield different trade-offs between component count and time. This example

is based on bit-slice multiplier components available in both the Schottky TTL and MECL 10,000 logic families [AMD 1, Motorola 1974].

Another function that requires fewer TTL integrated circuits to implement is the barrel shift that is used to normalize floating-point numbers and to do floating-to-integer and integer-to-floating conversions. A barrel shifter has data inputs and control inputs. The control inputs specify the number of places the data inputs are to be shifted when they appear at the output. In a floating-point addition, the two fractions to be added must have the same exponent. To accomplish this, the fraction with the smaller exponent is shifted right by an amount equal to the difference of the two exponents. To perform an integer-to-floating conversion, a priority encoder (available in either family) is used to find the position of the first one-bit in the integer representation. This position determines both the exponent of the floating-point representation and the number and direction of the shifts to produce the fractional part of the floating-point number. In a floating-to-integer conversion, the exponent specifies the amount the fraction must be shifted to produce the desired integer. Figure C-3 shows that type conversions are done in several places by independent units. The lack of a barrel shifter is a significant disadvantage of the MECL 10,000 family. Of course, a shifter can be constructed from other logic functions, but this results in a substantial

increase in the number of components required to perform the shift. It is also possible to implement a barrel shift function in an ECL system by converting ECL signals to TTL signals (there are components that do this), using the TTL barrel shifters, and finally converting back to ECL signals.

So the choice between Schottky TTL and MECL 10,000 is not clear-cut. The estimates here assume the use of MECL 10,000 components. The prices used are almost all from the Motorola Semiconductor Products Incorporated price list issued in February 1976 [Motorola 1976]. Motorola is the originator and major manufacturer of emitter-coupled logic. The prices are for quantities between 100 and 999; more than 1000 integrated circuits are used.

Memory components make up a substantial fraction of the components used in the high-performance ladder processor. In Figure C-2 there are two types of memory, memory for constants and memory for temporaries. The size of the temporaries is determined by the word size of the host machine. The cost estimates are based on the choice of a PDP-10 for the host computer. The word length of both the constant memory and the temporary memory is then 36 bits plus bits for type information and parity. The memory for the two copies of the temporaries contains 78 integrated circuits. The MCM10144L, a 256x1 random-access memory with a typical access time of eighteen nanoseconds, is used for this memory; it costs $19.50. The constant memory contains

64 words of 38 bits each. As Figure C-2 shows, random-access memory rather than read-only memory is used for this storage. The flexibility of the random-access memory is necessary only during the debugging of the processor. Table 5-1 lists the memory requirements for the high-performance ladder processor.

The computational and type conversion elements contain over half the components used in the processor. Each of the three integer-to-floating conversion units contains roughly 120 integrated circuits and can perform a conversion in about forty nanoseconds. The most straightforward method to implement the 36-bit integer multiply contains 630 integrated circuits and can produce the 72-bit product in less than 100 ns. But a full 72-bit product is not required, so the component count of the multiplier can be reduced to about 200. The smaller multiplier produces a 36-bit product in roughly 100 ns. Table 5-2 gives the breakdown of the components in all the arithmetic units.

The majority of the components in Figures C-2 and C-3 not listed in Tables 5-1 or 5-2 are data selectors. These components are used extensively in every section of the processor. Table 5-3 gives the approximate number of each type of multiplexer used. The use is not broken down into functional units because these components are used almost everywhere.

Use	Size	Component	Number	Unit Cost	Cost
Temporaries	2x256x39	MCM10144L	78	$19.50	$1521.00
Constants	64x38	MCM10140L	38	17.00	646.00
Subscript	512x24	F10415A	24	34.08	817.92
DELTA	512x24	F10415A	24	34.08	817.92
RHO	512x24	F10415A	24	34.08	817.92
G	512x24	F10415A	24	34.08	817.92
PI	64x24	MCM10140L	24	17.00	408.00
BETA	64x24	MCM10140L	24	17.00	408.00
PC	64x13	MCM10140L	13	17.00	221.00
TYPE	64x5	MCM10140L	5	17.00	85.00
RANK	64x4	MCM10140L	4	17.00	68.00
CRJREG	64x7	MCM10140L	7	17.00	119.00
uPC Stack	64x13	MCM10140L	13	17.00	221.00
uPGM Memory	1024x85	F10415A	85	34.08	2896.80
Code Memory	4096x22	F10415A	88	34.08	2999.04
Subtotals			475		$12,864.52

The F prefix of the component identification denotes that the manufacturer is Fairchild Semiconductor Components Group, Fairchild Camera and Instrument Corporation.

The MCM prefix indicates that the manufacturer is Motorola Semiconductor Products Incorporated.

Memory Requirements for the High-Performance Processor

Table 5-1

Use	Component	Number	Unit Cost	Cost
Integer ADD, SUB, and Logical	MC10181P	9	$14.08	$126.72
	MC10179P	3	2.80	8.40
Integer Multiply	MC10287L	189	14.38	2717.82
	MC10181P	19	14.08	267.52
	MC10149L	21	43.50	913.50
	MC10179P	1	2.80	2.80
	MC10176P	3	4.00	12.00
Integer Divide	MC10181P	9	14.08	126.72
	MC10179P	3	2.80	8.40
	MC10173P	18	2.84	51.12
Integer-to-Floating (3 units)	MC10164P	279	2.82	786.78
	MC10165P	18	5.63	101.34
	MC10149P	6	43.50	261.00
Floating-to-Integer	MC10164P	92	2.82	259.44
	MC10181P	7	14.08	98.56
	MC10179P	3	2.80	8.40
Floating-Point unit	MC10287L	351	14.38	5047.38
	MC10164P	125	2.82	352.50
	MC10181P	16	14.08	225.28
	MC10173P	21	2.84	59.64
	MC10179P	6	2.80	16.80
	MC10165P	5	5.63	28.15
Other Arithmetic Units	MC10181P	24	14.08	337.92
	MC10179P	8	2.80	22.40
Subtotals		1,236		$11,840.59

The MC prefix in the component number indicates that the manufacturer is Motorola Semeconductor Products Incorporated.

Components in Arithmetic Units of the High-Performance Processor

Table 5-2

Roughly ten per cent of the components in the ladder processor are not in Tables 5-1, 5-2, or 5-3. These components have little impact on the cost of the processor and are ignored.

Component	Number	Unit Cost	Cost
MC10164P	137	$2.82	$386.34
MC10173P	74	2.84	210.16
MC10174P	39	2.82	109.98
Subtotals	250		$706.48
Grand totals	1,961		$25,411.59

Data Selectors/Multiplexers

Table 5-3

Tables 5-1, 5-2, and 5-3 show that the high-performance ladder processor design outlined in Appendix C contains more than 2000 integrated circuits and that the cost of these is over $25,000.

Tables 5-4a and 5-4b give the corresponding component usage for the moderate-performance ladder processor.

Some of the multiplexers shown in Figure D-1 are not included in Table 5-4. These devices are not needed because the data selection is done by means of a tri-state bus. They were shown in Figure D-1 to illustrate the processor's logical operation, not its physical implementation.

Use	Organization	Component	Number	Unit Cost	Cost
Constant Memory	16x33	SN74S189N	9	$ 6.65	$ 59.85
uPGM Memory	4096x60	F93448DC	64	24.70	1580.80
Memory	4096x33	F93415ADC	132	18.20	2402.40
Subtotals			205		$4,043.05

Memory Components in the Moderate-Performance Ladder Processor

Table 5-4a

Use	Component	Number	Unit Cost	Cost
Microprocessor	Am2901DC	8	$42.00	$336.00
Look Ahead Carry	SN74S182N	3	4.50	13.50
uPGM Sequencer	Am2909DC	3	25.96	77.88
MAR Mux	SN74S153N	6	4.12	24.72
Carry In Mux	SN74S251N	2	5.09	10.18
Branch Condition Mux	SN74S251N	2	5.09	10.18
Subtotals		24		472.46
Grand Totals		229		$4,515.51

Arithmetic and Data Selection Units in the Moderate-Performance Ladder Processor

Table 5-4b

The SN prefix indicates that the manufacturer is Texas Instruments Incorporated.

The F denotes Fairchild Semiconductor Components Group, Fairchild Camera and Instrument Corporation.

The Am prefix indicates that the manufacturer is Advanced Micro Devices Incorporated.

The memory for the code and data requires more integrated circuits than any other part of the system. The component used for this memory is the Fairchild 93415A 1024x1 random-access memory [Fairchild 1975]. There are TTL compatible 4096x1 RAMs that would reduce the number of components in this memory by a factor of four, but they are slow. The access time of a typical fast 4K RAM is 200 ns while the access time of the 93415A is thirty ns. It is very likely tht a faster 4K RAM will become available in 1976, so that, if the moderate-performance ladder processor were to be built several months in the future, these fast 4K RAMs would be the most appropriate integrated circuits for the memory. It is likely that the price of these components will be only slightly more than the price of the 1K RAMs in Table 5-4a.

The total cost of the components for the moderate-performance ladder processor is roughly $4,500. The new 4K RAMs would allow this cost to be reduced to about $3,000. A similar reduction in price (with an accompanying loss of performance) is possible with 4K RAMs currently available.

5.2 PERFORMANCE OF THE TWO DESIGNS

Making a performance estimate for either processor is complicated. The execution time for single instructions can be estimated, but the performance measure of interest is the execution time of APL programs. Two ALGOL programs were

written to enable such an estimate. The first translates an APL statement into code for the ladder processor and the second simulates the ladder processor. As part of the simulation, the program counts the instructions as it executes them. The instruction count and the execution time of the instructions give an estimate of the execution time of an APL statement. Appendix E contains the translation program and Appendix F contains the simulation program.

The propagation delays of the components in the processors determine the minimum instruction-execution times. The instruction-execution times for the high-performance ladder processor determined by this method vary from 100 ns for branch and coroutine jumps to 150 ns for integer add and subtract instructions and iteration-step instructions. The execution time of an integer multiply instruction is about 200 ns and of an integer divide is 1200 ns. Floating-point additions, subtractions, and multiplications require roughly 200 ns while floating-point divisions require 1000 ns. The microprograms for some instructions contain several microinstructions. The execution time of an individual microinstruction, except multiply and divide, is about 100 ns. The execution time of the first microinstruction in a microprogram is about fifty ns longer because the time required by phases normally overlapped with the previous microinstruction must be included.

The microinstructions of the moderate-performance ladder processor require about 120 ns. The execution time of each macroinstruction is determined by the number of microinstructions in the microprogram for the instruction. An instruction that adds two temporaries executes in about 720 ns, an iteration-step instruction takes about 1700 ns, and a coroutine jump about 1200 ns. The microinstruction field that suspends the processing of microinstructions until the memory has completed the current operation is not necessary if the processor uses the Fairchild 93415A components in the memory. These memory components have a typical access time of 30 ns; this time can be absorbed in the 120 ns microinstruction cycle. If slower, cheaper memories were used, the memory access time might be 200 to 250 ns. This time cannot be absorbed in the instruction cycle. If a microinstruction references the output of the memory, the execution time of the previous microinstruction must be extended until this output is ready. A processor with the slower memory would run about half as fast as a processor with the 30 ns memory.

Breed and Lathwell gave estimates for the execution time of simple APL statements using an IBM 360/50 [1967]. The setup time is between two and three milliseconds and the execution time is between forty and 250 microseconds per element. Of course, without further explanation it's impossible to guess the most complicated "simple" statement they had in mind, but the simplest is probably is A←B. The

execution time for the 360/50 implementation plotted in Figure 5-1 is derived from the smallest time estimates given by Breed and Lathwell.

A ladder network for A←B is the same for the two ladder processors. There are six instructions in the innermost loop -- two iteration-step instructions, two coroutine jumps, and two MOVEs.

The high-performance ladder processor executes this code in about 800 ns. This time can be attained only if the memory of the host machine can perform a read and a write operation in 800 ns. The main memory of most computers that might be used as the host machine in a high-performance APL system cannot support this operation rate. The cycle times of the main memory of many medium and large computers today are around one microsecond. If the high-performance ladder processor were connected to a host computer with such a slow memory, the innermost loop of the instruction A←B would take at least two microseconds. If the memory of the host computer were four-way interleaved, the high-performance ladder processor would use one quarter of all memory cycles for loop execution times that are close to two microseconds. This size load on the memory of a time-shared computer might not be acceptable; certainly a greater load would not be. The ladder processor must be slowed down to lower the burden on the memory. Only for programs with complex operations like the trigonometric functions can the ladder processor

run at full speed. But the more complex operators are much less common than the simple ones.

The inner loop of the instruction A←B on the moderate-performance ladder processor contains the same instructions as the inner loop on the high-performance processor. The execution time of the loop is about eight microseconds. If the array contains 32-bit data and the word length of the host computer is sixteen bits, then four memory operations are performed in one pass through the loop. If the memory has a one-microsecond cycle time, these four cycles require half the available memory cycles.

There is no special hardware for floating-point operations or integer multiply in the moderate-performance processor as there is in the high-performance processor. As a result, the execution times of these instructions are several microseconds. The class of instructions whose execution time is longer than the cycle time of the main memory is much larger for the moderate-performance processor than for the higher performance processor. On systems where little interference with the host computer's use of the memory is allowed, the moderate-performance processor must be artificially slowed less often than the faster processor. If it is often necessary to slow down the moderate-performance processor to reduce its use of the host computer's memory, the design of the processor should be reviewed, and the denser, slower memory should be used for

the memory of the ladder processor.

The total time for a ladder-based APL system to execute an APL statement is the sum of the time for the host computer to generate the ladder network and the time for the ladder processor to execute that network. The execution time can be computed by summing the execution times of all the instructions executed by the ladder processor. The ladder-network generation time is more difficult to estimate. When the host computer first encounters an APL statement, it must check the conformability of the variables in the statement. If the statement is legal, it must then generate a ladder network for the statement. The host machine can save the network so that the next time it encounters the statement, it need not regenerate the ladder network. It can use the previously generated network for the statement if none of the ranks or types of the variables have changed. In the previous discussion of the division of tasks between the host machine and the ladder processor, conformability checking was assigned to the host machine, so that, even if the ranks and types of the variables in an APL statement remain fixed, each time the host machine encounters the statement, it must perform a conformability check before proceeding with the execution.

The checking of the rank and type and the conformability of the arrays in a simple APL statement requires the execution of roughly 200 instructions on the

host computer. Another 100 or 200 instructions are sufficient to load the data and code into the ladder processor. 400 is a conservative estimate of the number of host machine instructions that must be executed before the ladder processor can begin executing the network for a simple APL statement. A moderately powerful host computer can execute this many instructions in about one millisecond. Therefore, if it is not necessary to regenerate the ladder network, the setup time of the ladder processor for a simple APL statement is about a half to a third the setup time for the software interpreter described by Breed and Lathwell [1967].

An estimate of the setup time is more difficult if the ladder network must be generated before processing of the APL statement can continue. There are two ways to arrive at such an estimate.

First, the work necessary to generate a ladder network is similar to the setup done by a software interpreter. The code generation of the ladder network is more elaborate than the generation of the internal representation of the APL statement used by the APL interpreter. An estimate that ladder-network generation requires fifty per cent more time than the setup of a standard APL interpreter is probably high.

The second way to arrive at an estimate is independent of the first. The program given in Appendix E generates ladder networks from an APL statement. This program was not written to run efficiently. The program was written in ALGOL because ALGOL is well suited to generating ladder networks and because the ALGOL system is probably the fastest language processor on the PDP-10. A measurement of the execution time of this program does not provide a realistic estimate of the execution time of a similar program in a production environment. The algorithm used in the generator program is not inefficient per se. The emphasis of the implementation was on making the writing and debugging of the program easy, allowing measurements of the program's performance to be made simply, and ensuring that the program was organized in an easily understandable way. Using the algorithm of the generator and estimating the execution time of an efficient implementation of the steps in it can provide a realistic estimate of the time required to generate a ladder network. This approach results in an estimate that fewer than 1500 instructions are required to generate a simple ladder network. This estimate is in rough agreement with the estimate made using the first method.

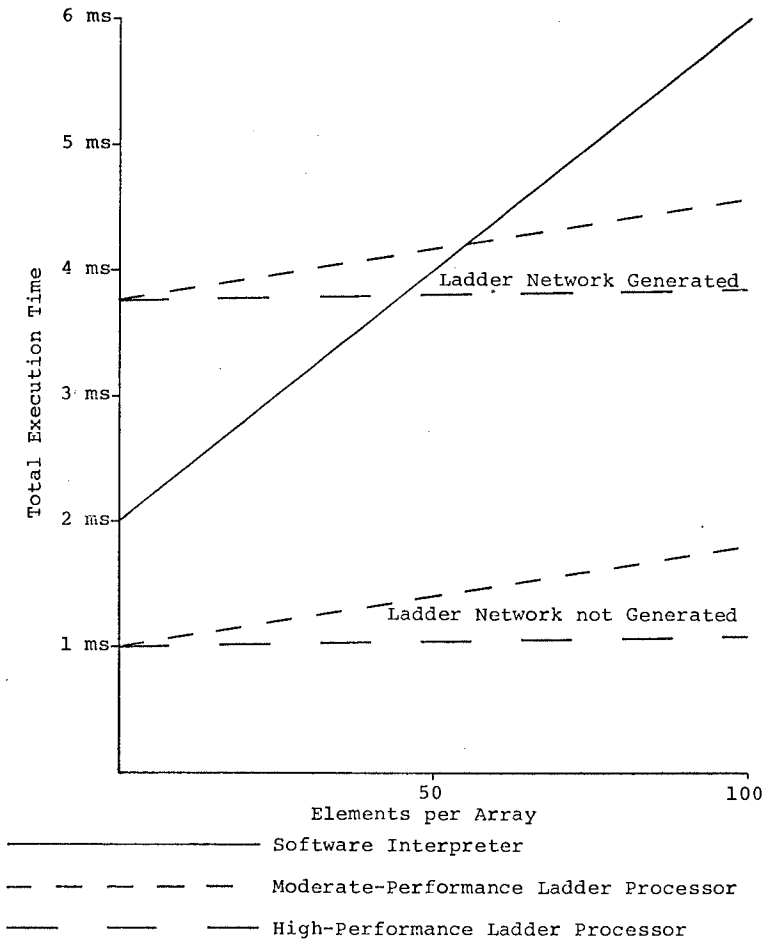
There is little difference between the number of instructions required to generate a ladder network for the two ladder processors. The code generator for the moderate-performance processor is more complicated because type conversion between integer and floating-point

representations is not automatic in that processor. But this results in only a small increase in code generation time.

The results of all the performance estimates are presented in Figure 5-1. The vertical axis is the time for the execution of an APL statement and the horizontal axis is the number of elements in the arrays in the statement. None of the estimates of the performance of the two-processor APL system consider that the host computer can process other tasks while the ladder processor executes the ladder network. The effectiveness of this parallelism depends primarily on the operating system and the job mix in the host computer.

The program given in Appendix F simulates the action of the ladder processor as it executes a ladder network. As the program steps through the code of the network, it maintains a count of the number of instructions executed. The product of this count and the average instruction-execution time is the time required for the ladder processor to execute the ladder network. The total time to execute an APL statement is the sum of the time to generate a ladder network and the time to execute it.

The Greater Washington APL User's Group ran three benchmark APL programs on several computers to compare the performance of the APL systems [Dusold 1975]. One of the benchmark programs tested branching. The ladder processor



Execution Time of A←B

Figure 5-1

works with only one statement at a time and must be reloaded by the host machine to process the next statement. The program that simulates the ladder processor provides no information about branching. Another of the benchmark programs is intended to test the floating-point arithmetic library. This test could be run on the simulator but it does not provide much information beyond the quality of the implementation of floating-point arithmetic. The third program (+/(⍳200)∈⍳200), which checks the ability to handle simple array operations, was used to test the performance of both ladder processors. Table 5-5 contains the results of these tests and the original test data.

The simulations of the execution of the ladder network for the statement indicates that the execution would take about nine milliseconds on the high-performance ladder processor and about 65 milliseconds on the slower processor. The time to generate the network would be less than ten milliseconds. The times given in the reference are in sixtieth's of a second; Table 5-5 gives these same times in milliseconds (ms). The reference gives two times for the IBM 360/91 -- 233 ms for APL.SV and less than 4 ms for APL.SV modified for the model 91 by the staff of the Applied Physics Laboratory of Johns Hopkins University. The faster of these two times is better than the time for the high-performance ladder processor by a factor of five. The fast time for the 360/91 is the fastest time by far of all the times in the survey. The Univac 1108 time was 50 ms;

Computer	Program Product	Execution Time (Milliseconds)
IBM 370/168	APL.SV	283
IBM 370/158	APL.SV	433
IBM 370/158	XM6	567
IBM 360/91	APL.SV	233
IBM 360/91	APL.SV (Modified, see text)	4
DEC PDP-10	APL.SS	500
Burroughs B7700	APL/700	133
UNIVAC 1108	UMD APL	50
High-Performance Ladder Processor		19
Moderate-Performance Ladder Processor		75

(Data for existing machines and programs from Dushold [1975].)

Execution Times for +/(⍳200)∈⍳200

Table 5-5

half as fast as the high-performance ladder processor. The IBM 370/158 and the PDP-10 had times in the neighborhood of half a second, more than six times slower than the moderate-performance ladder processor.

These timing comparisons are more favorable for the ladder processors than the comparisons with the IBM 360/50 that used a different method of comparison because the ladder mechanism maintains subscripts for arrays, and the subscript for a dummy vector can be used to produce the monadic iota function. Another consequence of this method of implementing the monadic iota function is that no references to the memory of the host computer are required during the execution of the ladder network. This means that the ladder processor is not slowed down by a slow main memory, and that the host computer is not slowed down by competition for its memory.

This evaluation indicates that an APL system using either of the ladder processors would perform better than many APL systems except those based on the most powerful computers. The performance comparisons in this chapter do not take into account the fact that the ladder-based systems have two processors, the CPU of the host machine and the ladder processor. Since these machines can run in parallel the effective performance of the system is higher than the figures given here indicate. The advantages of parallelism are especially important for the moderate-performance

processor because it is cheap enough so that even a reasonably small computer system can include more than one.

5.3 COMPARISON OF THE TWO LADDER PROCESSORS

The costs of the processors estimated earlier in the chapter were for the integrated circuits only. There are other costs, both direct and indirect, involved with building either of the processors. Any comparison of the costs of the processors should be based on the total cost, not just the cost of the logic circuitry.

The task of completing the design of the high-performance ladder processor and constructing a prototype is immense. The total cost of the components is at least $50,000. This cost includes power supplies, racks, printed-circuit boards, and so on. At least during the first part of debugging, work on the ladder processor will probably tie up the host machine. Since the host computer is a large system, this disruption is expensive. Building a prototype of the high-performance ladder processor would take three to four man-years of effort. Few installations can justify and afford such large investments of time and money.

On the other hand, the total component cost for the moderate-performance ladder processor varies between $5,000 and $10,000 depending on the price of the memory components used. The development work can be done on a much smaller

host computer system than the system on which the processor will eventually run. The development time of the system would be about one man-year. The architecture of the moderate-performance processor is flexible enough so that the designer can change the characteristics of the processor substantially by changing only a few parts of the processor (the microprograms and the instruction decoding logic). Finally, the solid-state devices that will be introduced in the next few years will probably benefit the moderate-performance processor more than the higher-performance processor. Chapter 6 discusses the impact of trends in the semiconductor industry on the two processors in more detail.

The ratio of the performance of the two ladder machines is about ten to one. But as Figure 5-1 shows, the time required to load the ladder network for a statement into the ladder processor can be a large fraction of the total execution time of the statement. For this reason the performance of an APL system based on the high-performance processor will be considerably less than a factor of ten times the performance of a system with the slower processor.

The costs of the machines are fluid but it appears that the high-performance processor will cost between five and ten times as much as the moderate-performance processor. Thus, the price/performance ratios of the two processors are close.

CHAPTER 6

CONCLUSION

Both of the APL processors discussed in Chapter 5 offer very good performance compared to most of the software APL systems for which Dusold [1975] gave timings. The high-performance processor is much faster than the moderate-performance one, but it is also less flexible and much more expensive. Any work on the implementation of a ladder processor should concentrate on the lower-performance processor because it is much less demanding to implement.

The APL system described in this work contained two processors -- a general-purpose CPU and the ladder processor. There are two reasons to use this organization. First, the processors run in parallel and so have higher throughput than either one of the processors running alone. The ladder processor, because of its more restricted duties, can be simpler than the CPU and should, as a result, be cheaper for the same performance. The second reason for using two processors is that executing an APL program naturally breaks into two parts: a supervisory task and a computing task. Other APL systems, based on software [Breed 1967], on firmware [Grant 1974], and on hardware [Abrams 1970], have also used two processors.

The use of a general-purpose computer and slave processor system is not limited to APL. It is useful in any situation when a small amount of code and data is used in a

lengthy computation. The reason is obvious.

The code and data used in the computation should be stored in a multiported or separate memory so that accesses do not interfere with the computer's use of its own memory. The slave processor's references to the host computer's memory are infrequent. The host computer must load the instructions and data into the slave processor. The time it takes for the host computer to load a simple instruction into the slave processor is roughly the same as the time it takes the slave to execute it. If each instruction is executed only once, none of the host computer's time was saved. But if the code contained loops with large repeat counts, then the slave processor saves a lot of the host computer's time. The loops can be explicit, as they are in the ladder processor, or implicit, as they would be for a processor with a complicated instruction set. The amount of the host computer's time saved is directly proportional to the loop repeat counts.

It is possible to substantially reduce the work the host machine must perform in loading the memory of the slave processor by organizing the slave's memory as a cache connected to the memory of the host machine. This organization is more difficult to implement and can place restrictions on how the host machine allocates its memory.

APL is ideally suited to a two-processor implementation because arrays are the fundamental data type. This approach could also be used for a SNOBOL system where string matching is a common but time-consuming operation.

The components used to implement the processors described here are changing so rapidly that an evaluation of the processors' cost and performance is not complete without a prediction of the impact of these changes. Since the introduction of integrated circuits about fifteen years ago, the number of components per chip has almost doubled each year, and it is likely that this rate will continue at least until 1980 [Hodges 1976]. During the same fifteen year period the maximum speed of these circuits has gone up by a factor of four, and this trend also is expected to continue. The different rates of development of the speed and density have strongly influenced the evolution of computers. The fastest development has been in the equipment that uses the moderate-performance, high-density integrated circuits -- first the minicomputer and now the microcomputer. Since semiconductors will continue to evolve as they have in the past, the rapid development of lower cost, lower performance computers will also continue. The effect of this trend on the moderate-performance ladder processor will be large -- it is likely that there will be an improvement of a factor of ten in the price/performance ratio by 1980. The improvement in the high-performance processor will be much less dramatic. There are indications that the speeds of the

current integrated circuits are close to the best possible for silicon devices [Keyes 1975, Hodges 1976]. It is possible to use another technology to build logic circuits, but any major shift is years off.

The impact of this trend in the development of semiconductor devices extends beyond ladder processors. Moderate-performance computers will continue to become cheaper both in an absolute sense and relative to larger computers. Some of the implications of this continued shift are clear. An analysis based on hardware trends alone emphatically suggests that distributed computing, where it is feasible, will be more cost effective than centralized computing.

APPENDIX A

ARRAY ADDRESSING

It is sometimes necessary to reference a particular element of an array using the subscripts of that element. For an address-sequencing mechanism to be suitable for general use, it must be possible to calculate the address of an element from its subscripts without a great deal of difficulty. The following discussion shows how this can be done for the address-generation mechanism discussed in Chapter 2.

An obvious way to compute the address of a particular element within the structure discussed in Chapter 2 is to begin with the first element and cycle the address-generation mechanism through all the elements of the array in ravel order until the desired element is reached. While this technique is clearly inefficient, it does provide a way to derive a more acceptable computational procedure.

Suppose that the desired element is $A[J_1;J_2;\ldots J_n]$ and that we have stepped through the array in ravel order, element by element, until we arrived at the point where PI points to this element. Refer to Figure 2-1 and consider the decision box for the first or outermost subscript. The only exit from this box that has been used is the one selected when $I_1 < RHO_1$. When the exit corresponding to

$I_1 > RHO_1$ is selected the ladder has sequenced through the entire array and the subscript combination $[J_1;J_2;\ldots J_n]$ was not found and was therefore illegal. But let us assume that the subscript combination is legal. Each time that the $I_1 < RHO_1$ exit of the decision box is taken, the code in the box in the horizontal path that this exit leads to is executed. Since the first subscript starts with a value of 1 and is incremented by 1 each time the $I_1 < RHO_1$ exit is taken, after control has passed through the decision box (and thus the horizontal box corresponding to it) $J_1 - 1$ times the first subscript will have value J_1.

At this point the remaining problem is exactly analogous to the original one except there are now $N - 1$ subscripts. This problem's I_2 corresponds to the original problem's I_1 and by a similar argument to the one above the path where I_2 is incremented must be traversed $I_2 - 1$ times in addition to the number of times it was traversed in the original problem. To compute the number of times the horizontal path corresponding to the second subscript is traversed during the first part of the problem (while I_1 is being stepped to J_1) consider the decision box associated with the second subscript. The $I_2 \geq RHO_2$ exit is taken once and the $I_2 < RHO_2$ exit is taken $RHO_2 - 1$ times; therefore, the number of traversals of the horizontal path associated with the second subscript during the first part of the problem is $(J_1 - 1)(RHO_2 - 1)$. So for the

k-th subscript the number of times its corresponding horizontal path is traversed is $J_k - 1$ plus $RHO_k - 1$ times the sum of the number of times all the horizontal paths associated with subscripts I_1 through $I_k - 1$ have been traversed while those subscripts were being stepped to the values J_1 through $J_k - 1$.

$$T_k = J_k - 1 + (\rho_k - 1) \sum_{i=1}^{k-1} T_i \qquad (1)$$

T_i is the number of traversals of the horizontal path associated with the i-th subscript. A simple induction argument will show that (1) is equivalent to

$$T_k = J_k - 1 + (\rho_k - 1) \sum_{i=1}^{k-1} (J_i - 1) \prod_{\ell=i+1}^{k-1} \rho_\ell . \qquad (2)$$

(1) and (2) are equivalent if and only if

$$\sum_{i=1}^{k-1} T_i = \sum_{i=1}^{k-1} (J_i - 1) \prod_{\ell=i+1}^{k-1} \rho_\ell . \qquad (3)$$

For $k = 2$ we have $T_1 = J_1 - 1$ which is the value of T_1 given by (1). Now we add T_k, as given by (1), to both sides of (3):

$$T_k + \sum_{i=1}^{k-1} T_i = T_k + \sum_{i=1}^{k-1} (J_i - 1) \prod_{\ell=i+1}^{k-1} \rho_\ell$$

$$\sum_{i=1}^{k} T_i = J_k - 1 + (\rho_k - 1) \sum_{i=1}^{k-1} T_i + \sum_{i=1}^{k-1} (J_i - 1) \prod_{\ell=i+1}^{k-1} \rho_\ell$$

$$= J_k - 1 + (\rho_k - 1)\sum_{i=1}^{k-1}(J_i - 1)\prod_{\ell=i+1}^{k-1}\rho_\ell$$

$$+ \sum_{i=1}^{k-1}(J_i - 1)\prod_{\ell=i+1}^{k-1}\rho_\ell$$

$$= J_k - 1 + \rho_k \sum_{i=1}^{k-1}(J_i - 1)\prod_{\ell=i+1}^{k-1}\rho_\ell$$

$$= J_k - 1 + \sum_{i=1}^{k-1}(J_i - 1)\prod_{\ell=i+1}^{k}\rho_\ell$$

$$= \sum_{i=1}^{k}(J_i - 1)\prod_{\ell=i+1}^{k}\rho_\ell$$

Since (3) is true for $k = 2$ and since we have just shown that if it is true for $k - 1$ it is true for k, then (3) must be true for all $k \geq 2$. Therefore, (1) and (2) are equivalent.

On each traversal of the k-th horizontal path $DELTA_k$ is added to PI so that:

$$LOC(A[J_1;J_2;\ldots;J_N]) = \beta + \sum_{i=1}^{N}\delta_i T_i$$

$$= \beta + \sum_{i=1}^{N}\delta_i[J_i - 1 + (\rho_i - 1)\sum_{k=1}^{i-1}(J_k - 1)\prod_{\ell=k+1}^{i-1}\rho_\ell] \qquad (4)$$

Using (4) we can compute the location of an array element given its subscripts. Since the addressing scheme ladders use is similar to FORTRAN's (though it is more general as discussed in Chapter 2), it is reasonable to expect that

the expressions that give the location of an array element as a function of subscript values are similar. In that case we would expect an expression of the form

$$LOC(A[J_1;\ldots;J_N]) = \beta + \sum_{k=1}^{N} G_k(J_k - 1) \qquad (5)$$

where G_k are expressions we must find.

From (4) and (5) we see that we would like to find G_i such that

$$\sum_{i=1}^{N}(J_i - 1)G_i = \sum_{i=1}^{N}\delta_i[J_i - 1 + (\rho_i - 1)\sum_{k=1}^{i-1}(J_k - 1)\prod_{\ell=k+1}^{i-1}\rho_\ell].$$

Let $G_i = \delta_i + \sum_{k=i+1}^{N}(\rho_k - 1)\delta_k \prod_{\ell=i+1}^{k-1}\rho_\ell$. (6)

We would like to show:

$$\sum_{i=1}^{N}(J_i - 1)[\delta_i + \sum_{k=i+1}^{N}(\rho_k - 1)\delta_k \prod_{\ell=i+1}^{k-1}\rho_\ell]$$

$$= \sum_{i=1}^{N}\delta_i[J_i - 1 + (\rho_i - 1)\sum_{k=1}^{i-1}(J_n - 1)\prod_{\ell=k+1}^{i-1}\rho_\ell] \; . \qquad (7)$$

For $N = 1$ we have $(J_1 - 1)\delta_1 = (J_1 - 1)\delta_1$. The left-hand side of (7) with N replaced by $N + 1$ becomes:

$$\sum_{i=1}^{N+1}(J_i - 1)[\delta_i + \sum_{k=i+1}^{N+1}(\rho_k - 1)\delta_k \prod_{\ell=i+1}^{k-1}\rho_\ell]$$

$$= \sum_{i=1}^{N}(J_i - 1)[\delta_i + \sum_{k=i+1}^{N+1}(\rho_k - 1)\delta_k \prod_{\ell=i+1}^{k-1}\rho_\ell]$$

$$+ (J_{N+1} - 1)\delta_{N+1}$$

$$= (J_{N+1}-1)\delta_{N+1} + \sum_{i=1}^{N}(J_i-1)[\delta_i + \sum_{k=i+1}^{N}(\rho_k-1)\delta_k \prod_{\ell=i+1}^{k-1}\rho_\ell + (\rho_{N+1}-1)\delta_{N+1}\prod_{\ell=i+1}^{k-1}\rho_\ell]$$

$$= (J_{N+1}-1)\delta_{N+1} + \sum_{i=1}^{N}(J_i-1)(\rho_{N+1}-1)\delta_{N+1}\prod_{\ell=i+1}^{k-1}\rho_\ell + \sum_{i=1}^{N}(J_i-1)[\delta_i + \sum_{k=i+1}^{N}(\rho_k-1)\delta_k \prod_{\ell=i+1}^{k-1}\rho_\ell] \quad (8)$$

The right-hand side of (7) with N replaced by N + 1 becomes:

$$\sum_{i=1}^{N+1}\delta_i[J_i-1+(\rho_i-1)\sum_{k=1}^{i-1}(J_k-1)\prod_{\ell=k+1}^{i-1}\rho_\ell]$$

$$= \delta_{N+1}[J_{N+1}-1+(\rho_{N+1}-1)\sum_{k=1}^{N}(J_k-1)\prod_{\ell=k+1}^{N}\rho_\ell] + \sum_{i=1}^{N}\delta_i[J_i-1+(\rho_i-1)\sum_{k=1}^{i-1}(J_k-1)\prod_{\ell=k+1}^{i-1}\rho_\ell. \quad (9)$$

We wish to show that (8) and (9) are equivalent. By assuming that (7) is true for N we see that the last term in (8) is equal to the last term in (9). After deleting these terms we wish to show that

$$(J_{N+1}-1)\delta_{N+1} + \sum_{i=1}^{N}(J_i-1)(\rho_{N+1}-1)\delta_{N+1}\prod_{\ell=i+1}^{k-1}\rho_\ell = \delta_{N+1}[J_{N+1}-1+(\rho_{N+1}-1)\sum_{k=1}^{N}(J_k-1)\prod_{\ell=k+1}^{N}\rho_\ell]. \quad (10)$$

By removing constant factors in the summation in the left-hand side of (10) we obtain

$$(J_{N+1}-1)\delta_{N+1} + \delta_{N+1}(\rho_{N+1}-1)\sum_{i=1}^{N}(J_i-1)\prod_{\ell=i+1}^{k-1}\rho_\ell$$

and finally

$$\delta_{N+1}[J_{N+1}-1+(\rho_{N+1}-1)\sum_{i=1}^{N}(J_i-1)\prod_{\ell=i+1}^{k-1}\rho_\ell] = \delta_{N+1}[J_{N+1}-1+(\rho_{N+1}-1)\sum_{i=1}^{N}(J_i-1)\prod_{\ell=i+1}^{k-1}\rho_\ell] .$$

Since (7) is true for N = 1 and since we have shown that if it is true for N then it must be true for N + 1, then (7) must be true for all $N \geq 1$. Therefore we have $LOC(A[J_1;J_2;\ldots;J_N]) = \beta + \sum_{i=1}^{N} G_i(J_i-1)$ where the G_i are given by (6).

The expression in (6) for G_i may be simplified

$$G_i = \delta_i + \sum_{k=i+1}^{N}(\rho_k-1)\delta_k\prod_{\ell=i+1}^{k-1}\rho_\ell$$

$$= \delta_i + \sum_{k=i+2}^{N}(\rho_k-1)\delta_k\prod_{\ell=i+1}^{k-1}\rho_\ell + (\rho_{i+1}-1)\delta_{i+1}$$

$$= \delta_i + \sum_{k=i+2}^{N}(\rho_k-1)\delta_k\,\rho_{i+1}\prod_{\ell=i+2}^{k-1}\rho_\ell + (\rho_{i+1}-1)\delta_{i+1}$$

$$= \delta_i + \rho_{i+1}\sum_{k=i+2}^{N}\delta_k(\rho_k-1)\prod_{\ell=i+2}^{k-1}\rho_\ell + (\rho_{i+1}-1)\delta_{i+1}$$

$$= \delta_i + \rho_{i+1}(G_{i+1}-\delta_{i+1}) + (\rho_{i+1}-1)\delta_{i+1}$$

$$G_i = \delta_i + \rho_{i+1}G_{i+1} - \delta_{i+1} \cdot \qquad (11)$$

From (6) we see that $G_N = \delta_N$ and from that point (11) gives the remainder of the G_i. The following ALGOL program fragment computes the G from the DELTA, RHO, and the rank.

```
G[RANK]:=DELTA[RANK];
FOR I:=RANK - 1 STEP -1 UNTIL 1 DO
  G[I]:=DELTA[I] + RHO[I+1] * G[I+1] - DELTA[I+1];
```

Suppose that a set of G_i are known such that (5) gives the address of an element form its subscripts. Is there a corresponding set of δ_i such that the ladder mechanism can be used to step through the array in ravel order? To answer this question first note that G_k represents the difference in addresses of two elements adjacent in the k-th dimension.

$$\mathrm{LOC}(A[J_1;J_2;\ldots;J_k;\ldots;J_N])$$

$$= \beta + \sum_{i=1}^{N}(J_k - 1)G_i = \beta + \sum_{i=1}^{k-1}(J_i - 1)G_i + \sum_{i=k+1}^{N}(J_i - 1)G_i + (J_k - 1)G_k$$

$$\mathrm{LOC}(A[J_1;J_2;\ldots;J_k+1;\ldots;J_N])$$

$$= \beta + \sum_{i=1}^{k-1}(J_i - 1)G_i + \sum_{i=k+1}^{N}(J_i - 1)G_i + J_kG_k$$

Subtracting the upper equation from the lower yields

$$\mathrm{LOC}(A[J_1;\ldots;J_k;\ldots;J_N]) - \mathrm{LOC}(A[J_1;\ldots;J_k+1;\ldots;J_n]) = G_k$$

With this interpretation of the G_i it is immediately apparent that $G_N = \mathrm{DELTA}_N$ since they both represent the separation between elements adjacent in the N-th or innermost dimension. To find the relation between the remainder of the DELTA's and G's consider the problem of moving from element $A[J_1;J_2;\ldots;J_{k-1};J_k;\rho_{k+1};\ldots;\rho_N]$ to element $A[J_1;\ldots;J_{k-1};J_k+1;1;1;\ldots1]$. From the discussion above it is clear that the separation in addresses of these elements $G_k + \sum_{i=k+1}^{N}(\rho_i - 1)G_i$. The ladder mechanism would use a step of size δ_k to move from the first element to the second:

$$\delta_k = G_k + \sum_{i=k+1}^{N}(\rho_i - 1)G_i \qquad (12)$$

Now the question remains whether this method of computing the δ values from the values for G and ρ is compatible with the method derived above for computing the

G values from the values for δ and ρ (11).

$$\delta_{k-1} = G_k - \sum_{i=k}^{N} (\rho_i - 1)G_i$$

$$= G_{k-1} - \sum_{i=k+1}^{N} (\rho_i - 1)G_i - (\rho_k - 1)G_k$$

$$= G_{k-1} + G_k - \sum_{i=k+1}^{N} (\rho_i - 1)G_i - \rho_k G_k$$

$$\delta_{k-1} = G_{k-1} + \delta_k - \rho_k G_k$$

$$\text{or} \quad \delta_k = G_k + \delta_{k+1} - \rho_{k+1} G_{k+1} \qquad (13)$$

Equations (11) and (13) represent the same relationship; they are simply rearrangements of one another. Using the fact that $\delta_N = G_N$ (13) yields the following ALGOL program fragment to compute the δ values given the values for G and ρ.

```
DELTA[RANK]:=G[RANK];
FOR I:=RANK-1 STEP -1 UNTIL 1 DO
  DELTA[I]:=G[I] + DELTA[I+1] - RHO[I+1]*G[I+1];
```

Thus if an array is stored so that it can be accessed in ravel order by the ladder mechanism, it is possible to find a vector G such that the location of a particular array element, specified by its subscripts, is given by the following expression:

$$\text{LOC}(A[J_1;J_2;\ldots;J_N]) = \beta + \sum_{i=1}^{N} (J_i - 1)G_i \qquad (11)$$

Furthermore, we have shown that if there exists a vector G such that equation (11) always yields the correct location for an array element then a vector can be found that allows the array to be accessed in ravel order by the ladder machine.

APPENDIX B

SELECTION-OPERATOR TRANSFORMATIONS

As stated in Chapter 2 certain APL operators produce changes in data-access parameters (RHO, DELTA, G, and BETA) rather than data movement. Consequently these operators can be handled quite efficiently. The transformations in the access parameters these operators produce are derived below.

B.1 REVERSE

This operator produces the reversal of the elements along the specified dimension; for a vector the last element becomes the first, the next to last the second, and so on.

The G vector, derived in Appendix A, can be used with a specific set of subscripts to find the address of an element.

$$LOC(A[J_1;...J_N]) = BETA + \sum_{i=1}^{N} (J_i-1)G_i \qquad (1)$$

As shown in Appendix A, G_k is the separation between two elements with subscripts differing only in the k-th subscript and there by one. The application of a reverse operation on the k-th subscript does not change the separation of two array elements but it does change the sign of the separation along the k-th subscript.

$$G_k' = -G_k \qquad (2)$$

The base address of the array is also different because the element that was A[1;1;...1] is now A[1;1;...RHO_k;...1] where the RHO_k is the subscript in the k-th position. Therefore BETA becomes

$$\begin{aligned} BETA' &= BETA + G_k(RHO_k - 1) \\ &= BETA - G_k'(RHO_k - 1) \end{aligned} \qquad (3)$$

To verify that (2) and (3) produce the desired transformation consider the location of a particular element, $A[J_1;...J_k;...J_N]$. Since the data is not moved in storage, the location of this element before the reverse on the k-th subscript is performed becomes the location of the element $A'[J_1;...RHO_k-J_k+1;...J_N]$ after the reverse has been performed.

$$LOC(A[J_1;...J_N]) = LOC(A'[J_1;...RHO_k-J_k+1;...J_N])$$

$$BETA + \sum_{i=1}^{N} (J_i-1)G_i = BETA' + \sum_{i=1}^{k-1} (J_i-1)G_i + \sum_{i=k+1}^{N} (J_i-1)G_i + (RHO_k-J_k)G_k'$$

$$\begin{aligned} BETA + (J_k-1)G_k &= BETA' + (RHO_k-J_k)G_k' \\ &= BETA + G_k(RHO_k-1) - (RHO_k-J_k)G_k \end{aligned}$$

$$BETA + (J_k-1)G_k = BETA + (J_k-1)G_k$$

The old and new sets of subscripts do reference the same storage location, and, therefore, the same data element. Since the old and new sets of subscripts represent

the reverse operation the transformations (2) and (3) do produce the reverse along the k-th dimension. It is interesting to note that two reversals along the same dimension results is the original access parameters as, of course, they must.

As shown in Appendix A, the fact that the array resulting from the application of the reversal operator may be referenced by (1) implies that it can be accessed in ravel order by the ladder mechanism. The new DELTA vector can be computed by the program fragment given in Appendix A using the new G vector.

B.2 TAKE

Take is a dyadic operator whose right argument is an array and whose left argument is a vector of integers whose length is equal to the rank of the right argument. Each integer in the left argument corresponds to a coordinate of the right argument. If the integer corresponding to the k-th coordinate of the right argument, call it L_k, is greater than or equal to zero and less than or equal to the k-th element of the shape vector of the right-hand argument then the k-th element of the shape vector of the result is L_k. A legal set of subscripts for the resultant array references the same element (residing at the same address) of that array as the same set of subscripts applied to the original array. That is, the resultant array contains the first L_k elements along the k-th dimension of the original

array. If $-RHO_k \leq L_k < 0$ then the resultant array contains the last $|L_k|$ elements along the k-th dimension of the original array in the same order. Originally a value of L_k outside the range discussed above was illegal, but the definition of take has been extended to allow any value of L_k. For $L_k > RHO_k$ the k-th dimension of the resultant array is extended above by appending zeros (blanks for a character array) along the k-th dimension until the k-th element of the shape vector of the new array is L_k and for $L_k < -RHO_k$, zeros are appended below the first element until the total number of elements along the k-th dimension is $|L_k|$. Unfortunately the mechanism described below handles only the original take operator; the extended case can be handled by inserting code into the splices of the ladder set up to sequence through the right argument array. So, while both types of take operators can be handled within the ladder structure, only the original version can be handled by changing the data-access parameters.

When the take operator is applied to an array, the values of the separation between elements along each dimension of the array are not changed. The shape vector of the array is changed, and, as a result, the DELTA vector is also changed. The absolute value of an element of the left argument is the number of elements in the new array along that dimension. The shape vector of the new array is simply the vector of the absolute values of the left argument. The new values of DELTA can be computed using the program

fragment given in Appendix A.

If every element of the left argument is non-negative or equal to minus the corresponding element of the shape vector of the right-hand argument then BETA does not change. This is easy to see because under these conditions the first element in the array (the element with all subscripts equal to 1) does not change. If any element of the left argument is negative and not equal to the negative of the corresponding element of the shape vector of the right argument then the address of the first element of the array does change and BETA must be changed accordingly.

Suppose that all the elements of the left argument are equal to the corresponding element of the shape vector of the right argument except for L_k, the k-th element; this means that except for the k-th dimension the array is not changed. If $L_k<0$ and $L_k \neq -RHO_k$ then the first $RHO_k-|L_k|$ elements are deleted from the k-th dimension of the right argument array.

$$LOC(A[1;...RHO_k-|L_k|+1;...1]) = LOC(A'[1;...1])$$

The subscript $RHO_k-|L_k|+1$ occurs in the k-th subscript position.

$$BETA' = BETA + (RHO_k-|L_k|)G_k$$

The effects of negative valued elements of the left argument are independent so we may write

$$BETA' = BETA + \sum_{i=1}^{N} (RHO_i-|L_i|)G_i(L_i<0)$$

$$BETA' = BETA + \sum_{i=1}^{N} (RHO_i+L_i)G_i(L_i<0)$$

The logical expression takes on the value 1 when the relation is true and 0 otherwise.

The storage used by an array which is the result of the application of the take operator on another array is not contiguous (except for certain combinations of the left argument and the shape of the right argument). The holes in storage correspond to sets of subscripts which were legal when used with the old array but which are not legal when used with the new array. If the subscripts used as inputs to (1) are legal, no references will be made to the holes in the storage of the new array even though the values for G remain unchanged. The new values of DELTA which can be computed from the values for G and the new values for RHO will cause the address sequence generated by the ladder mechanism to skip over these holes. The following program fragment performs the take operation:

```
FOR I:= 1 UNTIL RANK DO
BEGIN
  IF TAKEVEC[I]<0 THEN
     BETA:=BETA+(RHO[I]+TAKEVEC[I])*G[I];
  RHO[I]:=ABS(TAKEVEC[I]);
END;
```

RANK,BETA, and RHO are the rank, BETA, and RHO of the right argument and TAKEVEC is the left argument.

B.3 DROP

The drop operator is the complement of the take operator. When the same arguments are used with the two operators, the drop operator deletes the elements of the right argument which the take operator retains and vice versa. The shape vector of the resultant array is the shape vector of the right argument minus the absolute value of the left argument vector. The new base address is given by

$$BETA' = BETA + \sum_{i=1}^{N} L_i G_i \ (L_i > 0)$$

The following ALGOL program fragment performs a drop:

```
FOR I:=1 UNTIL RANK DO
BEGIN
  IF DROPVEC[I]>0 THEN BETA:=BETA+DROPVEC[I]*G[I];
  RHO[I]:=RHO[I]-ABS(DROPVEC[I]);
END;
```

The variable naming convention is the same as for the take program fragment.

B.4 TRANSPOSE

The left argument of transpose is a vector whose length is equal to the rank of the right argument. The elements of the vector specify the new positions for subscripts of the array; the monadic transpose operator simply interchanges the last two coordinates of the argument. The transpose operator has no effect on BETA because the base element

remains the same. Often the left argument is a permutation of the integers 1 through the rank of the right argument; in this case the transpose can be accomplished by simply interchanging the elements of the G and RHO vectors as specified by the left argument. That is, the separation between elements along a particular coordinate and the shape vector element for that coordinate move unchanged with the coordinate subscript to the new position for that coordinate given by the left argument. For the monadic transpose operator the last two elements of the RHO and G vectors would be interchanged. Once the new values for RHO and G have been computed, the method derived in Appendix A can be used to compute new values for DELTA.

The requirement for the left argument is that its maximum element be less than or equal to the rank of the right argument, that all its elements be greater than zero, and that each integer between 1 and the maximum element be represented at least once. This means that some of the coordinates of the right argument can be mapped into the same coordinate position. In this case a step in the new coordinate is equivalent to a step in each of the coordinates which is mapped into it, and the shape vector component for the new coordinate is the minimum of the shape vector elements corresponding to each of the coordinates mapped into the new coordinate. The rank of the resultant array is the maximum component of the left argument. When several coordinates are mapped into the same coordinate the

component of the G vector for that coordinate is the sum of the components of the G vector corresponding to the mapped coordinates. An ALGOL program fragment to perform the transpose operation is given below.

```
FOR I:=1 UNTIL MAX DO
BEGIN
  MIN:=0;        GOUT[I]:=0;
  FOR J:=1 UNTIL RANK DO
  IF I=TVEC[J] THEN
  BEGIN
    MIN:=IMIN(MIN,RHOIN[J]);
    GOUT[I]:=GOUT[I]+GIN[J];
  END;
  RHOOUT[I]:=MIN;
END;
RANK:=MAX;
```

GIN and RHOIN are the G and RHO vectors for the right argument to the transpose operator, GOUT and RHOOUT are the G and RHO vectors for the result of the operation, and MAX is the largest component of the left argument. The variable RANK is equal to the rank of the right argument when the program fragment is entered; in the last statement it is assigned the value of the rank of the result of the transpose. Just as before the new values for DELTA may now be computed.

B.5 SUBSCRIPT

In order that the subscript operation be handled within the data-access mechanism provided by the ladder structure the subscript elements must be blank, a single constant, or an arithmetic progression. Certainly not all subscripts are one of the three types but these are common subscript types.

The subscripting is accomplished by computing vectors to be used as left arguments to take and drop operators applied to the argument array of the subscript operation. If the k-th subscript is blank then the k-th component of the vector used with the drop operator is zero and the k-th component of the vector used with the take operator is the k-th component of the shape vector of the array. If the k-th subscript is a constant, C, then the k-th component of the drop vector is C-1 and the k-th component of the take vector is 1.

The case where the k-th subscript is an arithmetic progression can be divided into four subcases. In the simplest of these the arithmetic progression is a sequence of numbers where each is one greater than the one before it. For this type of subscript the k-th component of the drop vector is one less than the first element of the arithmetic progression and the k-th component of the take vector is the length of the arithmetic progression. If the k-th subscript expression is of the form A-⍳B where A and B are integers such that all elements of this sequence are legal subscript values for the k-th coordinate, then a reverse on the k-th coordinate is first performed on the array then the k-th element of the drop vector is RHO_k-A+1 and the k-th element of the take vector is B.

The other two cases are generalizations of the first two; in the following two cases the step size in the arithmetic progression is not 1. If the k-th subscript expression is A+C×ι B with restrictions on A,B, and C similar to the ones above, the k-th component of the drop vector is A+C-1 and the k-th component of the take vector is C×B. After the drop and the take have been applied to the array the k-th component of the G vector is multiplied by C and the k-th component of the shape vector is divided by C. The final case is that of an arithmetic progression with negative, non-unit steps -- A-C×ι B. First a reverse on the k-th coordinate is applied to the array then the k-th element of the drop vector is RHO_k-A+C and the k-th component of the take vector is C×B. Next, just as in the case of the positive step, the k-th component of the G vector is multiplied by C and the k-th component of the shape vector is divided by C. The new values for DELTA cause the address sequencing mechanism to step C elements at a time in the k-th coordinate.

APPENDIX C

HIGH-PERFORMANCE LADDER PROCESSOR

This appendix contains a more detailed description of the high-performance ladder processor. The connection of this processor to the host computer was presented briefly in Chapter 3. The memory of the ladder processor was connected to the memory bus of the host computer just as any memory would be. This enables the host machine to load the data and the code for a ladder network into the ladder processor. The ladder processor accesses data in the memory of the host machine through a port in the multiport memory system which is dedicated to it.

The data-access pointers provide the addresses of all the references to the memory of the host machine. When the data-access pointer of a ladder is loaded, the next host memory location referenced by the ladder will very likely be the one pointed to by the value loaded into the pointer. Although, in some situations a ladder might make no reference to the host memory before its data-access pointer is reloaded, the best estimate of the next host memory location to be referenced by a ladder is the value most recently loaded into the data-access pointer for that ladder. The high-performance ladder processor can fetch data items from the host memory pointed to by the new values of the data-access pointer in anticipation that these data items will be referenced soon. In this way the array data

can be read from a memory with an access time of less than 100 nanoseconds (local RAM storage) rather than from a memory with an access time of 500 to 1000 nanoseconds (host memory). This predictive fetching can be handled by a logically separate unit. The block diagram of the high-performance APL system is shown in Figure C-1.

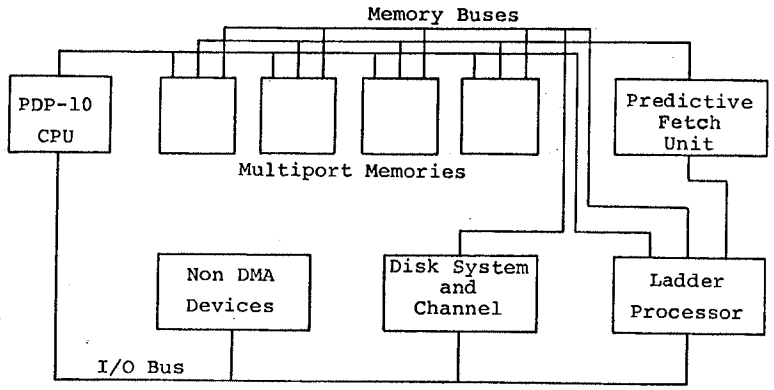
The instruction set for the ladder processor is given in Tables C-1, C-2, and C-4. Table C-3 gives the correspondence between the value of the operand field of double and single operand instructions and the data item referenced by that field. Table C-5 lists the microinstruction fields and their widths in bits; each field is assigned a field number and a name which clearly suggests its function. The component used as the arithmetic logic unit is the MC10181 [Motorola 1974]. The ALU FUNCTION SELECT signals are wired directly to the corresponding inputs of the ALU chips.

Tables C-6, C-7, and C-8 give sample microprograms. The microprograms in Table C-6 implement some of the simple ladder processor macroinstructions. The microprogram in Table C-7 implements the factorial operator (monadic !) for integer arguments. This microprogram detects a non-integer argument and branches to another microprogram (not given) to compute the value of the function for that type of argument. Table C-8 gives a micro-subroutine to compute the G vector from the RHO and DELTA vectors. Symbolic representations of

the two long microprograms are given in Chapter 4.

A block diagram of the logical organization of the high-performance ladder processor is given in Figures C-2 and C-3. The components diagramed in Figure C-2 perform the computational and data routing tasks associated with splice instructions. Figure C-3 illustrates the ladder data memory and the ladder code computational elements. Also, the micro and macro instruction sequencers and the coroutine link registers are shown in Figure C-3. Table C-9 gives logical expressions for some of the signals in Figures C-2 and C-3 which are not directly specified by a microinstruction field.

Chapter 5 contains estimates of the cost and performance of this ladder processor and the moderate-performance ladder processor described in Appendix D.

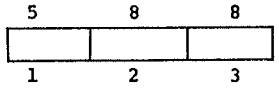


High-Performance APL System

Figure C-1

Instruction		Operation
LT	A,B	B←A<B
LE	A,B	B←A≤B
EQ	A,B	B←A=B
GE	A,B	B←A≥B
GT	A,B	B←A>B
NE	A,B	B←A≠B
OR	A,B	B←A∨B
AND	A,B	B←A∧B
NOR	A,B	B←A⍱B
NAND	A,B	B←A⍲B
XOR	A,B	B←(A∧B)∨(~A)∧(~B)
EQUIV	A,B	B←(A∧(~B))∨((~A)∧B)
ADD	A,B	B←A+B
MUL	A,B	B←A×B
SUB	A,B	B←B-A
RSUB	A,B	B←A-B
DIV	A,B	B←B÷A
RDIV	A,B	B←A÷B
RES	A,B	B←A|B
RRES	A,B	B←B|A
PWR	A,B	B←B*A
LOG	A,B	B←A⍟B
CIR	A,B	B←A○B
COMB	A,B	B←A!B
MAX	A,B	B←A⌈B
MIN	A,B	B←A⌊B
MOV	A,B	B←A
CMP	A,B	Sets Condition Codes Only

Instruction Format



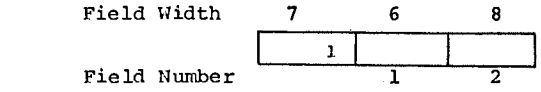
1 Op Code Field (No Double Operand Op Codes are Zero)
2 Source Operand Address Field
3 Destination Operand Address Field

Double Operand Instructions

Table C-1

Instruction		Operation
INC	A	A←A+1
DEC	A	A←A-1
TST	A	Sets Condition Codes Only
CLR	A	A←0
NEG	A	A←¯A
NOT	A	A←~A
RECIP	A	A←÷A
SIG	A	A←×A
CEL	A	A←⌈A
FLR	A	A←⌊A
ABS	A	A←|A
ROLL	A	A←?A
EXP	A	A←*A
LN	A	A←⍟A
FAC	A	A←!A
PIT	A	A←○A
SIN	A	A←1○A
COS	A	A←2○A
TAN	A	A←3○A
CIR4	A	A←4○A
SINH	A	A←5○A
COSH	A	A←6○A
TANH	A	A←7○A
CIR0	A	A←0○A
ASIN	A	A←¯1○A
ACOS	A	A←¯2○A
ATAN	A	A←¯3○A
CIRM4	A	A←¯4○A
ASINH	A	A←¯5○A
ACOSH	A	A←¯6○A
ATANH	A	A←¯7○A

Instruction Format



1 Op Code Field
2 Destination Operand Address Field

Single Operand Instructions

Table C-2

Address Field	Variable Referenced
0	the data item in the memory of the host machine where the data-access pointer points
1	data-access pointer
2	ladder program counter
3	base address of the array in the memory of the host machine
4	rank
5	type
6	origin (there is only one copy of the origin for the entire ladder network, not one for each ladder)
7	not used
8	first subscript
9	first component of the DELTA vector
10	first component of the shape vector
11	first component of the G vector
12	second subscript
13	second component of the DELTA vector
.	.
.	.
.	.
31	sixth component of the G vector
32	temporary 32
33	temporary 33
.	.
.	.
.	.
255	temporary 255

Temporaries 0 through 31 are shadowed by the ladder data. Only microinstructions (and the host computer) can reference these storage locations. They are used for accumulating intermediate results and temporary storage during the execution of the microprograms for the more complicated macroinstructions.

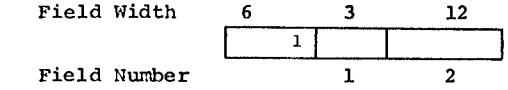
Address Correspondence Used in Single and Double Operand Instructions

Table C-3

Branch Instructions

BR LABEL	Branch always
BN LABEL	Branch never (NOP)
BNE LABEL	Branch if Z=0
BEQ LABEL	Branch if Z=1
BGE LABEL	Branch if N⊕V=0
BLT LABEL	Branch if N⊕V=1
BGT LABEL	Branch if Z∨(N⊕V)=0
BLE LABEL	Branch if Z∨(N⊕V)=1

Instruction Format

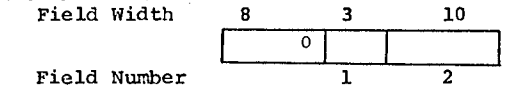


1 Branch Type
2 Offset

Iteration Step Instruction

ISTEP SUB,LABEL — Iteration Step for coordinate number SUB - 1
If there are more array elements along this subscript then branch to LABEL otherwise proceed to the next instruction.

Instruction Format



1 Coordinate Number + 1
2 Offset

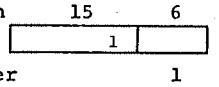
Miscellaneous Instructions

Table C-4 (Page 1 of 2)

Coroutine Jump Instruction

CRJ CRJREG — Swap contents of the Current Ladder Register (CURLAD) with the contents of Coroutine Link Register number CRJREG

Instruction Format

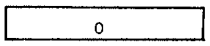


1 Coroutine Link Register Number

Halt Instruction

HALT — Stop execution of all instructions in the ladder processor and set the HALT flag to inform the host machine of the termination of ladder network execution.

Instruction Format



Miscellaneous Instructions

Table C-4 (Page 2 of 2)

Field Number	Field Name	Width
1	uPGM SOURCE TEMP ADR	8
2	uPGM SOURCE TEMP ADR MUX uPGM CONTROL	1
3	uPGM DEST TEMP ADR	8
4	uPGM DEST TEMP ADR MUX uPGM CONTROL	1
5	uPGM B LEG MUX CONTROL	3
6	uPGM B LEG MUX CONTROL SOURCE	1
7	uPGM B LEG MUX ENABLE	1
8	uPGM B LEG CONSTANT ENABLE	1
9	uPGM A LEG MUX CONTROL	3
10	uPGM A LEG MUX CONTROL SOURCE	1
11	uPGM A LEG MUX ENABLE	1
12	uPGM ALU FUNCTION SELECT MUX ADR	2
13	uPGM ALU FUNCTION SELECT MUX ENABLE	1
14	uPGM ALU FUNCTION SELECT (M,S0,S1,S2,S3)	5
15	uPGM MUL/DIV/REM SELECT	2
16	uPGM CARRY IN MUX ADR	3
17	uPGM STORE ENABLE (LDATA,TEMP)	2
18	uPGM FORCE A LEG TO FLOAT	1
19	uPGM FORCE B LEG TO FLOAT	1
20	uPGM ROUND MODE	2
21	uPGM LDATA SELECT	5
22	uPGM LDATA SELECT ENABLE	1
23	uPGM uBRANCH CONDITION MUX ADR	3
24	uPGM uBRANCH DESTINATION	12
25	uPGM PUSH uPC TO uPC STACK	1
26	uPGM POP uPC FROM uPC STACK	1
27	uPGM B LEG CONSTANT ADR	6
28	uPGM STOP MICROPROGRAM	1
29	uPGM OUTPUT BUS MUX ADR	2
30	uPGM OUTPUT BUS MUX CONTROL SOURCE	1
31	uPGM SWAP ALU INPUTS	1
32	uPGM FORCE INSTR DEST FIELD TO SOURCE TMP ADR	1

Microinstruction Fields

Table C-5

Field Number	GE		NOR	ADD	MUL	RSUB	RRES	MAX		MOV	CMP	INC	TST	NEG	SIG		CEL	FLR	ABS	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	1	1	1	1	1	1	1	0	1	1	1	0	1	0	1	1	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	3
13	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1
14	6	10	30	11	34	6	34	6	0	25	6	0	0	6	0	0	0	0	0	0
15	0	0	0	0	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	7	0	1	0	0	0	0	0	0	0	0	1	0	1	2	1	1	1	3
17	0	1	1	1	1	1	1	0	1	1	0	1	0	1	0	1	1	1	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	0	1
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	1	1	0	0	0	0	0	1	1	0	1	0	0	0	1	0	0	0	0
31	1	0	0	0	0	1	1	0	0	0	1	0	0	1	0	0	0	0	0	1
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(all fields in octal)

Examples of Microprograms for Macroinstructions

Table C-6

Field Number									
				LOOP			DONE	ZERO	
1	0	0	1	0	0	1	2	0	0
2	0	0	1	0	0	1	1	0	0
3	1	1	2	1	1	2	0	0	0
4	1	1	1	1	1	1	0	0	0
5	0	0	0	0	0	0	0	0	0
6	1	1	1	0	0	1	1	0	0
7	1	1	1	0	0	1	1	0	0
8	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0
10	0	1	1	1	1	1	0	0	0
11	0	1	1	1	1	1	0	0	0
12	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0
14	1	6	25	6	17	34	25	0	0
15	0	0	0	0	0	1	0	0	0
16	1	0	0	0	1	0	0	0	0
17	1	0	1	0	1	1	1	0	1
18	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0
20	1	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0
23	0	4	5	0	6	1	0	6	0
24	0	ZERO	GAMA	0	DONE	LOOP	0	GAMA	0
25	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	1	0	1
29	0	0	0	0	0	0	0	0	0
30	0	0	0	1	1	1	1	0	1
31	0	0	0	0	0	0	0	0	0
32	1	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Microprogram for Factorial of Integer Arguments

Table C-7

Field Number																	
				CONT													
1	0	0	0	2	0	0	0	0	0	0	0	0	0	2	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
3	1	1	2	0	1	2	1	2	1	2	1	2	2	0	2	2	2
4	1	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1
5	0	0	2	0	0	2	0	2	0	2	0	2	2	0	2	2	2
6	0	0	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1
7	0	0	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	1	1	0	0	1	0	1	0	1	0	1	0	0	0	1	1	1
11	1	1	0	0	1	0	1	0	1	0	1	0	0	0	1	1	1
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	17	17	25	25	17	25	17	25	17	25	17	25	25	25	34	11	6
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
16	1	1	0	0	1	0	1	0	1	0	1	0	0	0	0	1	0
17	1	1	1	2	1	1	1	1	1	1	1	1	1	2	1	1	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	4	0	11	13	0	15	0	21	0	25	0	31	35	37	36	31	35
22	1	0	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1
23	0	2	3	0	0	2	0	2	0	2	0	2	0	0	0	0	0
24	0	SCALAR	CONT	0	0	RNK2	0	RNK3	0	RNK4	0	RNK5	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Microprogram to Compute the G Vector from the RHO and DELTA Vectors

Table C-8 (Page 1 of 2)

Field Number	RNK5				RNK4				RNK3				RNK2				RNK1
1	2	0	0	0	2	0	0	0	2	0	0	0	2	0	0	0	2
2	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
3	0	2	2	2	0	2	2	2	0	2	2	2	0	2	2	2	0
4	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0
5	0	2	2	2	0	2	2	2	0	2	2	2	0	2	2	2	0
6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0	0
11	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	25	34	11	6	25	34	11	6	25	34	11	6	25	34	11	6	25
15	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
16	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
17	2	1	1	1	2	1	1	1	2	1	1	1	2	1	1	1	2
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	33	32	25	31	27	26	21	25	23	22	15	21	17	16	11	15	13
22	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Microprogram to Compute the G Vector from the RHO and DELTA Vectors

Table C-8 (Page 2 of 2)

```
TEMP WRITE = (uPGM TEMP STORE ENABLE)∧(INSTR WRITE PHASE)

SOURCE TEMP ADR MUX CONTROL 1 = (INSTR WRITE PHASE)∨
       (uPGM FORCE INSTR DEST FIELD TO SOURCE TEMP ADR)

SOURCE TEMP ADR MUX CONTROL 2 =
                        uPGM SOURCE TEMP ADR MUX uPGM CONTROL

SOURCE TEMP ADR MUX CONTROL 3 = HALT

SOURCE MUX ADR 1 = (uPGM SWAP ALU INPUTS)∧~(SOURCE MUX ADR 3)
SOURCE MUX ADR 2 =
      ((IRDECODE SOURCE LDATA REF)∧~(uPGM SWAP ALU INPUTS))∨
      ((IRDECODE DEST LDATA REF)∧(uPGM SWAP ALU INPUTS))

SOURCE MUX ADR 3 =
       ((IRDECODE @PI SOURCE)∧~(uPGM SWAP ALU INPUTS))∨
       ((IRDECODE @PI DEST)∧(uPGM SWAP ALU INPUTS))

DEST TEMP ADR MUX CONTROL 1 =
                uPGM DEST TEMP ADR MUX uPGM CONTROL

DEST TEMP ADR MUX CONTROL 2 = HALT

DEST MUX ADR 1 = (uPGM SWAP ALU INPUTS)∧~(DEST MUX ADR 3)

DEST MUX ADR 2 =
       ((IRDECODE DEST LDATA REF)∧~(uPGM SWAP ALU INPUTS))∨
       ((IRDECODE SOURCE LDATA REF)∧(uPGM SWAP ALU INPUTS)

DEST MUX ADR 3 =
       ((IRDECODE @PI DEST)∧~(uPGM SWAP ALU INPUTS))∨
       ((IRDECODE @PI SOURCE)∧(uPGM SWAP ALU INPUTS))
```

Signal Definitions

Table C-9 (Page 1 of 2)


```
A TO FLOAT = (A INT)∧((B FLOAT)∨(A FORCE)∨DIV∨
             (uPGM FORCE A LEG TO FLOAT)∨
             (B INT)∧((A FLOAT)∨(B FORCE)))

B TO FLOAT = (B INT)∧((A FLOAT)∨(B FORCE)∨DIV∨
             (uPGM FORCE B LEG TO FLOAT)∨
             (A INT)∧((B FLOAT)∨(A FORCE)))

FLOAT = (A FLOAT)∨(A TO FLOAT)∨(B FLOAT)∨(B TO FLOAT)

OUTPUT TO FLOAT =  ~FLOAT∧(FORCE OUTPUT)∧(IRDECODE @PI DEST)

OUTPUT TO INT = FLOAT∧(FORCE OUTPUT)∧(IRDECODE @PI DEST)

FLOAT OUT = FLOAT∧~(OUTPUT TO INT)∧~FLOAT∧(OUTPUT TO FLOAT)

OUTPUT TYPE ERROR = STORED∧(IRDECODE @PI DEST)∧
                          ((FLOAT OUT)≠(TYPE OUT))

OUTPUT BUS MUX CONTROL 1 = (OUTPUT TO FLOAT)∨(OUTPUT TO INT)

OUTPUT BUS MUX CONTROL 2 = FLOAT
```

Signal Definitions

Table C-9 (Page 2 of 2)

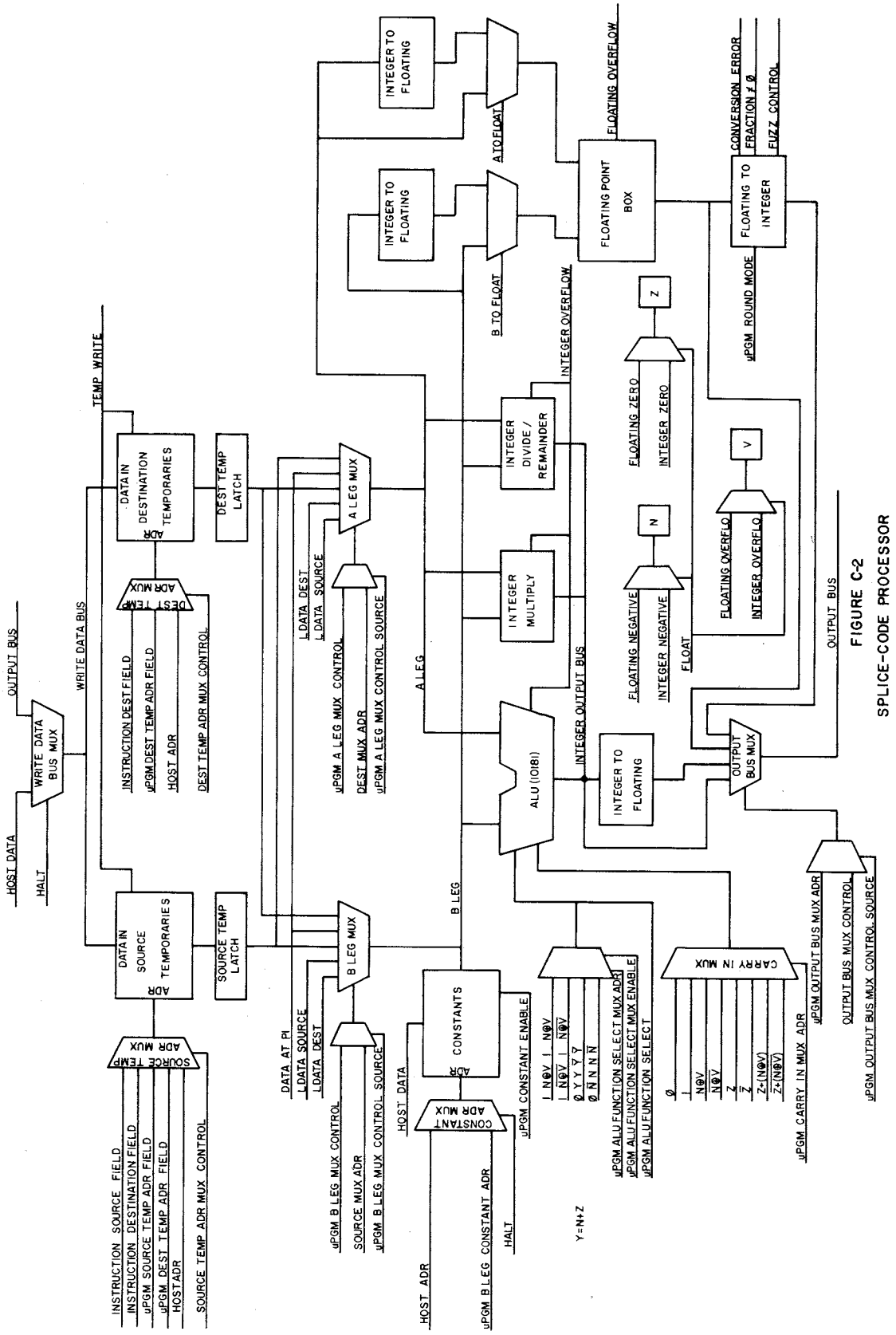


FIGURE C-2
SPLICE-CODE PROCESSOR

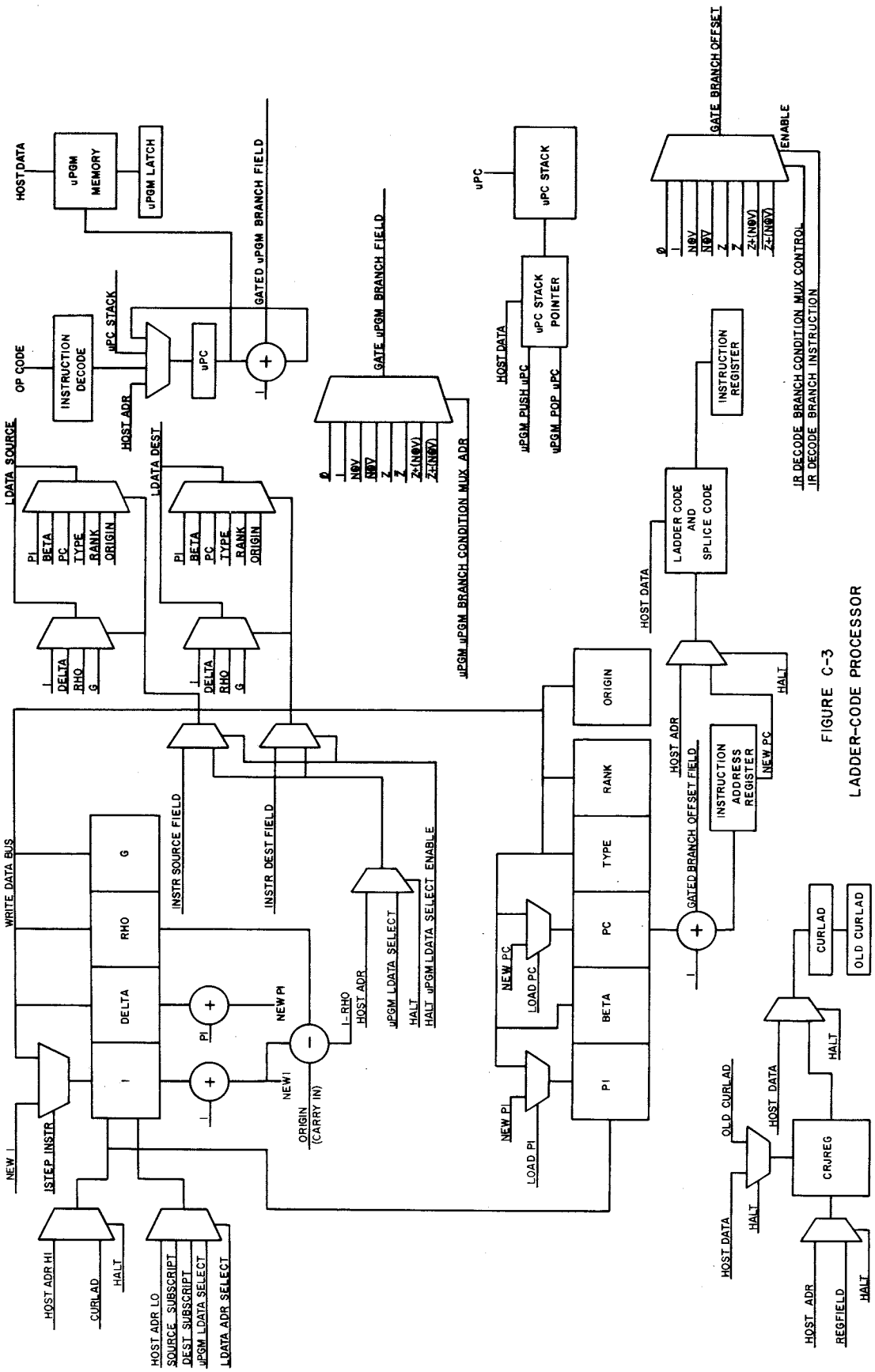


FIGURE C-3
LADDER-CODE PROCESSOR

APPENDIX D

MODERATE-PERFORMANCE LADDER PROCESSOR

As discussed in Chapter 4, the primary design goal of the moderate-performance ladder processor is low cost. Two general changes in the design described in Appendix C result in dramatic savings in both the number of components used and the cost. First, a single large memory is used to store all the instructions and data (except array data) used by the processor. Second, all computations in this processor are done with the same computational unit.

Memory components used in the moderate-performance ladder processor have much higher density than those used in the high-performance processor. This greater density enables the moderate-performance processor to store more than half as much data in less than one tenth the number of components used by the high-performance processor. (See Chapter 5 for a more detailed component count.) There are two reasons why the higher density memory components cannot be used in the high-performance processor. First, the organization of the dense memory chips (4096 x 1) is suitable only where a large number of data items, at least a few thousand, are to be stored. The data for the high-performance processor is stored in many independent memories so that several data items can be accessed concurrently. The sizes of these memories is small; some store 64 items, others store 256, and most of the rest store

512. The other reason is that the integrated circuit technology used in the high density memory components (MOS) is slow. The high-performance processor could execute several microinstructions in the time required for one complete cycle of a typical 4K MOS memory component. In fact, high density MOS memory with an organization more in line with that used in the high-performance ladder processor have recently been introduced for use with eight-bit MOS microprocessors. The use of these integrated circuits is ruled out soley because of their speed.

The savings achieved by using one common memory rather that several separate ones are more than just the savings in the number of memory components. A large number of multiplexers are used in the high-performance ladder processor to merge the outputs of the different memories into a single data path for use in computations done by the splice code. Of course, when all the data reside in the same memory these multiplexers are not needed.

The use of a single memory in the moderate-performance ladder processor implies that the construction of a memory address is more complicated than for the faster processor with its seperate memories. This address computation must be done under microprogram control in the one computational element contained in the processor. This burden can be removed by supplying a physical memory address in the operand address fields of an instruction instead of the

encoded, context sensitive (dependent upon CURLAD) addresses used in the faster processor.

With a few more components, the slower processor can form addresses like the faster processor. The minimum address size is six bits (32 ladder data and 32 temporaries) but seven bits is a more reasonable size (32 ladder data and 96 temporaries). If there are sixteen bits in the instruction, there could be a maximum of fifteen double operand instructions for six-bit adresses and three for seven-bit ones. It is clear that three double operand instructions are insufficient. If the instruction set of the ladder processor contained only fifteen double operand instuctions, it would be impossible to provide instructions for all the dyadic scalar operators. These operators would have to be implemented with a sequence of simpler instructions. From the example microprograms given below, it is clear that the bulk of the time required for the execution of a simple double operand instruction is spent decoding the instruction and loading the operands into registers internal to the arithmetic logic unit. Thus, a microprogrammed implementation of the more complicated operators is much faster than a software implementation. There are two ways to provide the large number of instructions required to efficiently implement the dyadic scalar operators. First, we can use a longer word. As discussed in Chapter 4, 32 bits is the logical choice. Second, we can decide not to have double operand

instructions, but to use a single address format or a memory-register format. The primary disadvantage of these instruction formats is not the fact that they are less expressive than double operand instructions; although this is true. The primary disadvantage is the additional logic required to process more than one instruction per instruction fetch. As pointed out in Chapter 4, an instruction size of 32 bits was selected chiefly to avoid this extra logic.

In the high-performance ladder processor, a spearate memory is used for each type of ladder data. References to a particular type must be decoded. As a result, little logic is required to add the unit to fetch array data from the host memory as soon as the data-access pointer is loaded with a new value. No such decoding is necessary in the slower processor because all ladder data is stored in the same memory. So, to add a unit to prefetch array data to this processor would require more logic than that required to add a similar unit to the faster processor. The moderate-performance ladder processor does not have this unit.

One advantage of using a physical memory address in double and single operand instructions is that the restriction on the maximum rank of an array can be removed. If the format of the iteration-step instruction is selected appropriately, the only limit on the rank of an array

results from the memory available to store the four values that must be stored for each coordinate. Since the hardware that prevented one ladder from referencing the ladder data of another ladder has been removed, it is the responsibility of the host machine to allocate the memory of the ladder processor so that no conflicts arise. But, as discussed previously, the removal of this restriction has little practical importance.

Following the organization used in Appendix C, the instruction set of the moderate-performance ladder processor is given in Tables D-1, D-2, and D-4. Table D-3 gives the order of the ladder data in memory. This particular order was selected to minimize the number of microinstructions in the microprogram for the iteration-step instruction. Table D-5 lists the microinstruction fields and their widths. The ALU CONTROL field is wired directly to the ALU control inputs of the Advanced Micro Devices 2901 four-bit bipolar microprocessor slice, the component used as the arithmetic logic unit and register file in this processor. The operation of this component as well as the microprogram sequencer (Am 2909) is described in a booklet distributed by the manufacturer [AMD 1975].

Table D-6 contains an explaination of the REGISTER LOAD AND FUNCTION CONTROL field and Table D-7 explains the uBRANCH CONDITION SELECT and the CARRY IN MUX SELECT fields. The fetch and decode microprogram is given in Table D-8.

Table D-9 lists other microprograms. A symbolic representation is also included in the tables.

The multiway branch is not described in any of the tables and its operation is rather involved. The multiway branch is the mechanism used to decode instructions. The contents of the instruction register is fed to a combinatorial logic circuit that performs two levels of decoding. First, it determines such general characteristics about the instruction as whether it is a single or double operand instruction and if either the source or destination operand is a reference to array data using the data-access pointer. In addition, the first level of decoding detects if the instruction is a HALT, CRJ, ISTEP, or any of the branch instructions. If the instruction is a branch, the logic further determines which branch it is. The FETCH microinstruction, shown in Table D-8, loads the instruction register with the contents of the memory location specified by the program counter and increments the program counter. At this point the instruction-decode logic determines the general class of the instruction. The multiway branch microinstruction that follows the FETCH instruction selects and branches to a different group of microcode to handle each case. All the instructions listed in Table D-4 are fully decoded so that, from this point on, the microprograms for them require no further decoding of the instruction register. The microprograms for the single and double operand instructions first fetch the operands. A second

level decode of the instruction register is performed and the microprogram for the instruction is selected. The moderate-performance ladder processor is much closer to a general-purpose machine than is its faster counterpart. The specialization of the moderate-performance processor is contained primarily in the instruction-decode logic and to a lesser degree in the data path organization.

Figure D-1 contains a block diagram of the logical organization of the moderate-performance ladder processor.

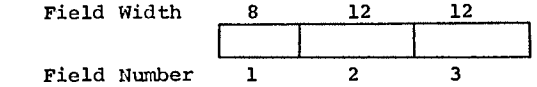
Instruction		Operation	
LT	A,B	B←A<B	The input operands are integers.
LE	A,B	B←A≤B	The output is the integer
EQ	A,B	B←A=B	representation of either 0 or 1.
GE	A,B	B←A≥B	
GT	A,B	B←A>B	
NE	A,B	B←A≠B	
FLT	A,B	B←A<B	The input operands are floating
FLE	A,B	B←A≤B	numbers.
FEQ	A,B	B←A=B	The output is the integer
FGE	A,B	B←A≥B	representation of either 0 or 1.
FGT	A,B	B←A>B	
FNE	A,B	B←A≠B	
OR	A,B	B←A∨B	These instructions perform
AND	A,B	B←A∧B	bit by bit logical functions.
NOR	A,B	B←A∨B	
NAND	A,B	B←A∧B	
XOR	A,B	B←(A∧B)∨(~A)∧(~B)	
ADD	A,B	B←A+B	The input operands and the
MUL	A,B	B←A×B	outputs of all these
SUB	A,B	B←B-A	instructions are integers.
RSUB	A,B	B←A-B	
DIV	A,B	B←B÷A	
RDIV	A,B	B←A÷B	
RES	A,B	B←A|B	
RRES	A,B	B←B|A	
FADD	A,B	B←A+B	The input operands and the
FMUL	A,B	B←A×B	outputs of all these
FSUB	A,B	B←B-A	instructions are floating
FRSUB	A,B	B←A-B	point numbers.
FDIV	A,B	B←B÷A	
FRDIV	A,B	B←A÷B	
FRES	A,B	B←A|B	
FRRES	A,B	B←B|A	

Double Operand Instructions

Table D-1 (Page 1 of 2)

MAX	A,B	B←A⌈B	
MIN	A,B	B←A⌊B	
FMAX	A,B	B←A⌈B	
FMIN	A,B	B←A⌊B	
PWR	A,B	B←B*A	The input operands and the
LOG	A,B	B←A⍟B	outputs of these instructions
CIR	A,B	B←A○B	are floating point numbers.
COMB	A,B	B←A!B	
IPWR	A,B	B←B*A	The input operands and the
ICOMB	A,B	B←A!B	outputs of these instructions are integers.
MOV	A,B	B←A	This instruction is independent of type.
CMP	A,B	Sets Condition Codes Only This instruction compares two integers.	
FCMP	A,B	Sets Condition Codes Only This instruction compares two floating-point numbers.	

Instruction Format



1 Op Code (No Double Operand Op Codes are zero.)
2 Souce Operand Address Field
3 Destination Operand Address Field

Double Operand Instructions

Table D-1 (Page 2 of 2)

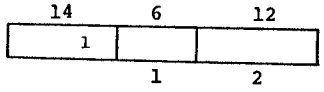
Instruction		Operation	
INC	A	A←A+1	These instructions take
DEC	A	A←A-1	integer arguments and
NEG	A	A←¯A	produce integer results.
ABS	A	A←|A	
ROLL	A	A←?A	
FAC	A	A←!A	
TST	A	Sets Condition Codes Only	
CLR	A	A←0	These instructions take
SIG	A	A←×A	either integer or floating point arguments and produce either integer or floating point results.
NOT	A	A←~A	This instruction takes a logical argument and produces a logical result.
CEL	A	A←⌈A	These instructions take
FLR	A	A←⌊A	floating arguments and
RND	A	A←⌊(A+.5)	produce integer results.
FLOAT	A	A←A	This instruction converts its argument from an integer to a floating-point number.

Single Operand Instructions

Table D-2 (Page 1 of 2)

Instruction		Operation	
FNEG	A	A←¯A	These instructions take floating arguments and produce floating results.
RECIP	A	A←÷A	
FABS	A	A←|A	
EXP	A	A←*A	
LN	A	A←⊛A	
FFAC	A	A←!A	
PIT	A	A←○A	
SIN	A	A←1○A	
COS	A	A←2○A	
TAN	A	A←3○A	
CIR4	A	A←4○A	
SINH	A	A←5○A	
COSH	A	A←6○A	
TANH	A	A←7○A	
CIR0	A	A←0○A	
ASIN	A	A←¯1○A	
ACOS	A	A←¯2○A	
ATAN	A	A←¯3○A	
CIR4	A	A←¯4○A	
ASINH	A	A←¯5○A	
ACOSH	A	A←¯6○A	
ATANH	A	A←¯7○A	

Instruction Format



1 Op Code Field
2 Destination Operand Address Field

Single Operand Instructions

Table D-2 (Page 2 of 2)

Location	Variable Referenced
0	data-access pointer
1	ladder program counter
2	base address of the array
3	rank
4	type
5	origin (there is one copy of the origin for each ladder in addition to one central copy)
6	first component of the shape vector
7	first subscript
8	first component of the DELTA vector
9	first component of the G vector
10	second component of the shape vector
11	second subscript
12	second component of the DELTA vector
13	third component of the shape vector
14	third subscript
.	.
.	.
.	.

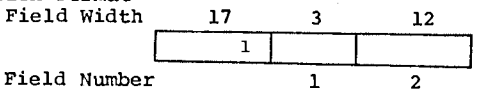
Location of the Ladder Data Relative to CURLAD

Table D-3

Branch Instructions

BR LABEL	Branch always
BN LABEL	Branch never (NOP)
BNE LABEL	Branch if Z=0
BEQ LABEL	Branch if Z=1
BGE LABEL	Branch if N⊕V=0
BLT LABEL	Branch if N⊕V=1
BGT LABEL	Branch if Z∨(N⊕V)=0
BLE LABEL	Branch if Z∨(N⊕V)=1

Instruction Format

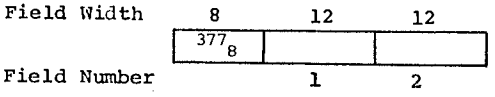


1 Branch Type
2 Offset

Iteration Step Instruction

ISTEP SUBOFF,LABEL — SUBOFF is the offset from CURLAD of the location of the four variable group for the coordinate tested by this instruction. If more iterations of this coordinate are to be done, a branch to LABEL is executed otherwise control passes to the next instruction.

Instruction Format



1 Subscript Group Offset
2 Branch Offset

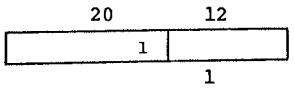
Miscellaneous Instructions

Table D-4 (Page 1 of 2)

Coroutine Jump Instruction

CRJ CRJLOC — Swap the contents of the current ladder register (CURLAD) with the contents of memory location CRJLOC.

Instruction Format

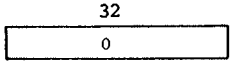


1 Coroutine Link Word Address

Halt Instruction

HALT — Stop the execution of all instructions by the ladder processor and set the HALT flag to inform the host machine host machine of the termination of the ladder network execution.

Instruction Format



Miscellaneous Instructions

Table D-4 (Page 2 of 2)

Field Number	Field Name	Width
1	uPGM A ADDRESS	4
2	uPGM B ADDRESS	4
3	uPGM ALU CONTROL	9
4	uPGM uBRANCH CONDITION SELECT	4
5	uPGM uBRANCH DESTINATION	12
6	uPGM D MUX SELECT	3
7	uPGM MAR MUX SELECT	2
8	uPGM WAIT FOR MEM DONE TO PROCEED	1
9	uPGM LOAD CONDITION CODES	1
10	uPGM CONSTANT ADR	4
11	uPGM CARRY IN MUX SELECT	4
12	uPGM MULTIWAY uBRANCH SELECT	1
13	uPGM MULTIWAY uBRANCH ENABLE	1
14	uPGM REGISTER LOAD AND FUNCTION CONTROL	4
15	uPGM INHIBIT OVERFLOW ERROR	1
16	uPGM PUSH uPC TO uPC STACK	1
17	uPGM POP uPC FROM uPC STACK	1
18	uPGM ENABLE MUL/DIV ALU CONTROL INPUTS	2

Microinstruction Fields

Table D-5

Field Value (octal)	Function
0	no action taken
1	load CURRENT TYPE REGISTER from OUTPUT BUS
2	load repeat count register from OUTPUT BUS
3	set FLOATING-POINT OVERFLOW ERROR
4	load MEMORY ADDRESS REGISTER from MAR MUX
5	load MEMORY ADDRESS REGISTER and initiate read cycle
6	load HOST MEMORY ADDRESS REGISTER from OUTPUT BUS
7	load HOST MEMORY ADDRESS REGISTER and initiate read
10	write memory from OUTPUT BUS
11	write host memory from OUTPUT BUS
12	initiate memory read cycle
13	initiate host memory read cycle
14	load MEMORY ADDRESS REGISTER, initiate read cycle, and load INSTRUCTION REGISTER from memory
15	load MEMORY ADDRESS REGISTER from MAR MUX and write memory from OUTPUT BUS
16	write last used memory from OUTPUT BUS
17	stop

Register Load and Function Control Field

Table D-6

Field Value (octal)	Branch Condition	Carry In
0	0 (Branch Never)	0
1	1 (Branch Always)	1
2	N⊕V	N⊕V
3	~(N⊕V)	~(N⊕V)
4	Z	Z
5	~Z	~Z
6	Z∨(N⊕V)	Z∨(N⊕V)
7	~(Z∨(N⊕V))	~(Z∨(N⊕V))
10	V	V
11	~V	~V
12	C	C
13	~C	~C
14	C∨Z	C∨Z
15	IRDECODE @PI DEST	~(C∨Z)
16	ORIGIN	ORIGIN
17	OVERFLOW ERROR	~ORIGIN

Branch Condition Select and Carry In Mux Select Fields

Table D-7

Field Number	Instruction Fetch and First Level Decode		Double Op Instr Op Fetch and 2nd Level Decode			Double Op Instr @PI Source Normal Dest				Double Op Instr Normal Source @PI Dest				Double Op Instr @PI Source @PI Dest			
1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	2
2	0	0	0	2	3	1	0	2	3	0	2	0	3	1	0	2	3
3	203	0	0	207	207	103	107	207	207	0	204	107	207	103	107	207	204
4	17	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
5	ERROR	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	2	0
7	3	0	1	2	0	3	0	2	0	1	3	0	0	3	0	0	0
8	1	0	1	1	0	1	1	1	0	1	1	1	0	1	1	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
13	0	1	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1
14	14	0	5	5	0	5	7	5	0	5	5	7	0	5	7	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Microprograms for Instruction Fetch and Decode

Table D-8 (Page 1 of 6)

AMD 2901 Internal Register Assignment

PC	0
CURLAD	1
SOURCE	2
DEST	3

```
FETCH:  MAR:=PC; PC:=PC+1; BC 17,ERROR; START MEM READ AND LOAD IR; WAIT
FTHP1:  IR DECODE 0

DOUBLE: MAR:=INSTR SOURCE FIELD; START MEM READ; WAIT
        SOURCE:=D; MAR:=INSTR DEST FIELD; START MEM READ; WAIT
        DEST:=D; IR DECODE 1

DOUBLE ^PI SOURCE: MAR:=CURLAD; START MEM READ; WAIT
        HMAR:=D; START HOST MEM READ; WAIT
        SOURCE:=D(@PI); MAR:=INSTR DEST FIELD; START MEM READ; WAIT
        DEST:=D; IR DECODE 1

DOUBLE @PI DEST: MAR:=INSTR SOURCE FIELD; START MEM READ; WAIT
        SOURCE:=D; MAR:=CURLAD; START MEM READ; WAIT
        HMAR:=D; START HOST MEM READ; WAIT
        DEST:=D(@PI); IR DECODE 1

DOUBLE @PI SOURCE @PI DEST: MAR:=CURLAD; START MEM CYCLE; WAIT
        HMAR:=D; START HOST MEM CYCLE; WAIT
        SOURCE:=D(@PI)
        DEST:=SOURCE; IR DECODE 1
```

Microprograms for Instruction Fetch and Decode

Table D-8 (Page 2 of 6)

Field Number	Single Op Instr		Single Op @PI Dest			HALT					COROUTINE JUMP							
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	3	1	0	3	1	0	0	1	0	1	0	0	1	1	0	0	0
3	0	207	103	107	207	103	103	147	103	103	103	103	0	207	103	207	105	107
4	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	FETCH
6	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	3	0
7	2	0	3	0	0	3	0	3	0	0	3	0	2	0	3	0	3	0
8	1	0	1	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0
11	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0
12	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
14	5	0	5	7	0	4	10	4	10	17	4	10	5	10	5	0	5	1
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Microprograms for Instruction Fetch and Decode

Table D-8 (Page 3 of 6)

AMD 2901
Internal Register
Assignment

PC	0
CURLAD	1
SOURCE	2
DEST	3

```
          SINGLE: MAR:=INSTR DEST FIELD; START MEM READ; WAIT
                  DEST:=D; IR DECODE 1

SINGLE @PI DEST: MAR:=CURLAD; START MEM READ; WAIT
                 HMAR:=D; START HOST MEM READ; WAIT
                 DEST:=D(@PI); IR DECODE 1

            HALT: MAR:=CURLAD+1
                  OUTPUT BUS:=PC; WRITE MEM; WAIT
                  MAR:=0
                  OUTPUT BUS:=CURLAD; WRITE
                  STOP

 COROUTINE JUMP: MAR:=CURLAD+1
                 OUTPUT BUS:=PC; WRITE MEM; WAIT
                 MAR:=INSTR DEST FIELD; START MEM READ; WAIT
                 CURLAD:=D; OUTPUT BUS:=CURLAD (OLD);
                            WRITE MEM; WAIT
                 MAR:=CURLAD+1; START MEM READ; WAIT
                 PC:=D
                 MAR:=CURLAD+4; START MEM READ; WAIT
                 TYPE:=D; BR FETCH
```

Microprograms for Instruction Fetch and Decode

Table D-8 (Page 4 of 6)

Field Number	ITERATION STEP											
1	1	0	0	0	5	0	0	0	0	0	7	0
2	4	5	4	6	6	0	6	4	7	1	0	0
3	305	207	303	207	121	0	103	103	207	103	105	205
4	0	0	0	0	0	6	0	0	0	0	0	1
5	0	0	0	0	0	FETCH	0	0	0	0	0	FETCH
6	4	0	0	0	0	0	0	0	0	0	0	1
7	3	0	3	0	0	0	0	3	0	3	0	0
8	1	0	1	0	0	0	1	1	0	1	0	1
9	0	0	0	0	1	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	1	0	14	0	1	1	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0
14	5	0	5	0	0	0	10	5	0	5	10	0
15	0	0	0	0	1	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Microprograms for Instruction Fetch and Decode

Table D-8 (Page 5 of 6)

AMD 2901
Internal Register
Assignment

PC	0
CURLAD	1
SOURCE	2
DEST	3
R4	4
R5	5
R6	6
R7	7

```
ITERATION STEP: R4,MAR:=CURLAD+INSTR SOURCE FIELD;
                       START MEM READ; WAIT
                R5:=D    !RHO;
                R4,MAR:=R4+1; START MEM READ; WAIT
                R6:=D    !SUBSCRIPT;
                OUTPUT BUS:=R5-R6-ORIGIN;
                       SET CONDITION CODES
                BLE FETCH
                OUTPUT BUS:=R6+1; WRITE MEM; WAIT
                MAR:=R4+1; START MEM READ; WAIT
                R7:=D;   !DELTA;
                MAR:=CURLAD; START MEM READ; WAIT
                OUTPUT BUS:=R7+D; WRITE MEM
                PC:=PC+INSTR DEST FIELD; WAIT; BR FETCH
```

Microprograms for Instruction Fetch and Decode

Table D-8 (Page 6 of 6)

Field Number	ADD	RSUB	BN	BR	BEQ		INC	NEG	ABS			SIG				
																MINUS
1	2	2	0	0	0	0	3	3	3	0	3	3	0	0	3	3
2	3	3	0	0	0	0	3	3	3	0	3	3	0	0	3	3
3	301	321	203	305	0	305	303	323	103	0	323	103	42	100	302	312
4	1	1	1	1	5	1	1	1	0	3	1	0	4	2	1	1
5	FETCH	FETCH	FTHP1	FETCH	FETCH	FETCH	FETCH	FETCH	0	FETCH	FETCH	0	FETCH	FETCH	MINUS	FETCH
6	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
7	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0
8	1	1	1	0	0	0	1	1	0	0	1	0	0	0	1	1
9	1	1	0	0	0	0	1	1	1	0	1	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	1	1	0	0	0	1	1	0	0	1	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	16	16	14	0	0	0	16	16	0	0	16	0	0	0	16	16
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(all fields except uBRANCH FIELD in octal)

Example Microprograms for Macroinstructions

Table D-9 (Page 1 of 2)

```
ADD:    OUTPUT BUS,DEST:=SOURCE+DEST; WRITE LAST USED MEMORY; LOAD CONDITION CODES; ENABLE OVERFLOW ERROR; WAIT; BR FETCH

RSUB:   OUTPUT BUS,DEST:=SOURCE-DEST; WRITE LAST USED MEMORY; LOAD CONDITION CODES; ENABLE OVERFLOW ERROR; WAIT; BR FETCH

BN:     MAR:=PC; PC:=PC+1; START MEM READ AND LOAD IR; WAIT; BR FTHP1;

BR:     PC:=PC+INSTR DEST FIELD; BR FETCH

BEQ:    BNE FETCH
        PC:=PC+INSTR DEST FIELD; BR FETCH

INC:    OUTPUT BUS, DEST:=DEST+1; WRITE LAST USED MEMORY; LOAD CONDITION CODES; ENABLE OVERFLOW ERROR; WAIT; BR FETCH

NEG:    OUTPUT BUS, DEST:=-DEST; WRITE LAST USED MEMORY; LOAD CONDITION CODES; ENABLE OVERFLOW ERROR; WAIT; BR FETCH

ABS:    OUTPUT BUS:=DEST; LOAD CONDITION CODES
        BPL FETCH
        OUTPUT BUS,DEST:=-DEST; WRITE LAST USED MEMORY; LOAD CONDITION CODES; ENABLE OVERFLOW ERROR; WAIT; BR FETCH

SIG:    OUTPUT BUS:=DEST; LOAD CONDITION CODES
        Q:=0; BEQ FETCH
        BMI MINUS
        OUTPUT BUS, DEST:=Q+1; WRITE LAST USED MEMORY; WAIT; BR FETCH
MINUS:  OUTPUT BUS, DEST:=Q-1; WRITE LAST USED MEMORY; WAIT; BR FETCH
```

Q is a scratch register in the AMD 2901.

Example Microprograms for Macroinstructions

Table D-9 (Page 2 of 2)

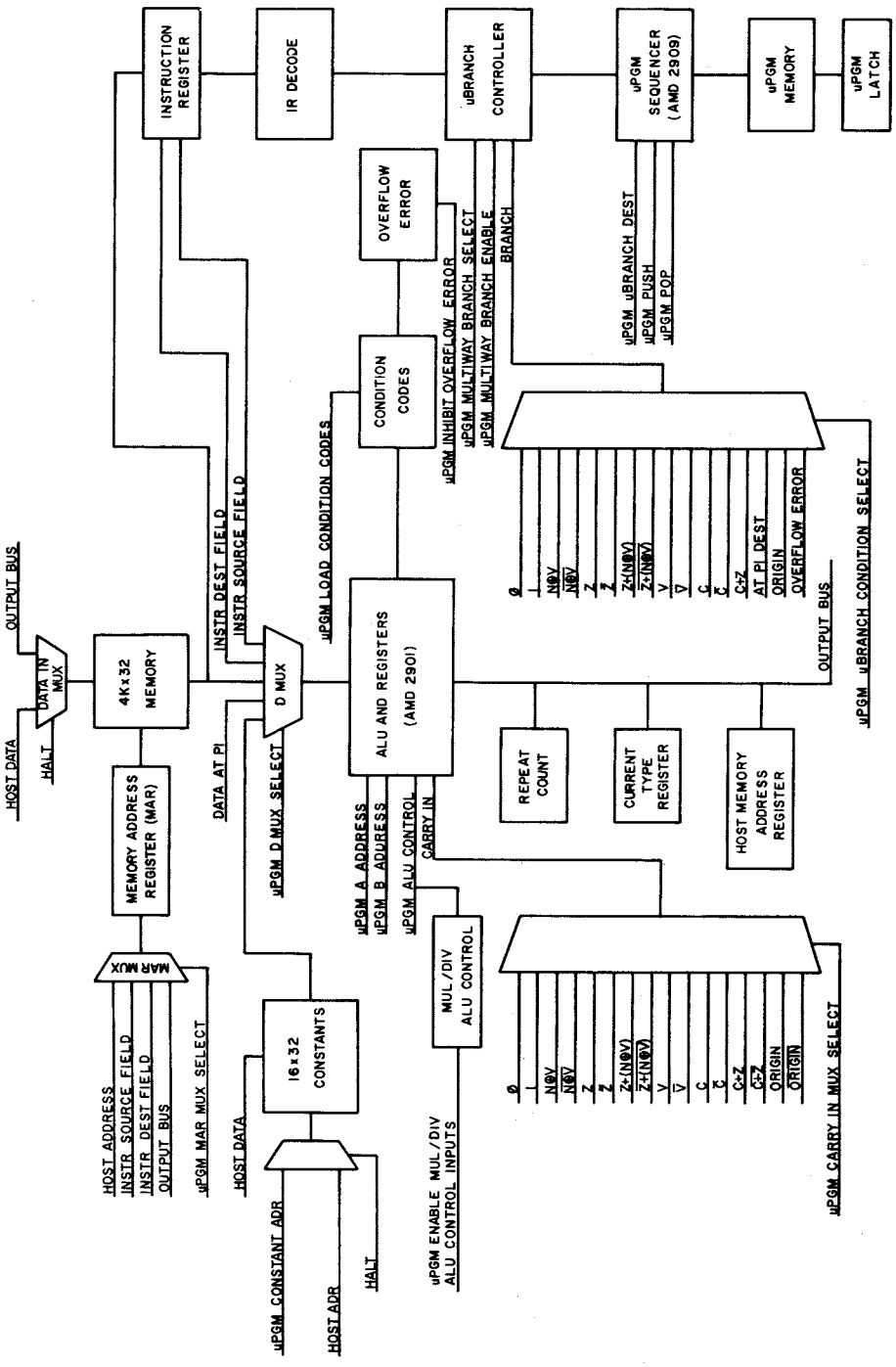


FIGURE D-1

MODERATE-PERFORMANCE LADDER PROCESSOR

APPENDIX E

APL TO LADDER NETWORK TRANSLATOR

The program in this appendix translates an APL statement in reverse Polish notation to a ladder network. The program in Appendix F can simulate the execution of the ladder network. Most, but not all, of the APL functions are implemented. Functions are divided into classes and all functions in the same class are handled by the same procedure. Only the dyadic and monadic scalar function classes contain more than one function. All other functions are handled by separate, usually small, procedures.

The symbols for all the functions are defined in a file that is read when the program begins. This file also defines for each monadic and dyadic scalar function the code to be generated, the identity element, and, for the dyadic functions, whether or not the function commutes. The translator does not handle some of the complicated monadic and dyadic scalar operators: for instance, the trigonometric functions. But these can be included by adding an entry to the function definition file; no change of the program is required.

Other functions not handled by the translator include deal, grade up, and grade down. These functions are not included because the ladder structure is not suited to their implementation. The program can perform the selection-operator transformations described in Appendix B.

These transformations assume that one or both of the operands are simple array references and not expressions. Abrams discusses methods to transform statements that do not satisfy these requirements into statements that do. Andrew Moulton is working on a program to transform statements like these into a format compatible with the program given here.

The program is slow; the translation of a complicated APL statement (about fifteen nodes in the parse tree) into a ladder network requires a few seconds. The program writes a file that lists all the input lexiems and, after each operator, the stack depth and the top frame of the stack.

The program prepares a file that contains splice code to be read by the program in Appendix F. All of this program's functions would be handled by the host processor in a two-processor system outlined in Chapter 3.

```
COMMENT   THIS IS FILE FT.ALG

THIS PROGRAM TRANSLATES A POLISH REPRESENTATION OF AN
APL STATEMENT INTO A LADDER REPRESENTATION.;

BEGIN   CHECKON;

  INTEGER NXTLAD,NXTTMP,NXTLREG,NOPRS,NOPNS,NAFB,OPRNUM,
          OPNNUM,FLDSIZE,MAXSINSTRS,INSTRFIELD,SPLFIELD,
           LADRFIELD,STARTLAD,MAXLADRS,MAXTEMPS,ILP,FLP,
          ASSGNOP,ACP,RCP,SCP,MAXOPNS,NWORDS,MAXWORDS,
          MAXRANK;
  BOOLEAN MOVCODE,ATPISOURCE,ATPIDEST,CRJCODE,SYMBMASK,
          CMPCODE,BNECODE,INCCODE,BINTYPE,INTTYPE,EXITMARK,
          TSTCODE,BEQCODE,RAVELCODE,CATENATECODE,
          COMPRESSCODE,EXPANDCODE,ENDMARK,BETASOURCE,
          MULCODE,ADDCODE,SOURCETL,DESTTL,LMASK,
          ALLOCTEMPOUT,IPIINSTR,DESTTR,SOURCEATPI,DESTATPI,
          SOURCETR,OPRMASK,OPNMASK,SOURCENEWTEMP,
          DESTNEWTEMP,DYADSCALMASK,MONADSCALMASK,
          HALTCODE,OUTPRODINSTR,REDUCINSTR,CLRCODE,
          PISOURCE,PIDEST,BETADEST,NOCOMMUTEMASK,SOURCET3,
          DESTT3,DESTT4,SOURCETIN,DESTT1,DESTT2,SCANINSTR,
          MBRSHPINSTR,INDXGENINSTR,
          INDXOFINSTR,REVERSEINSTR,SHAPEINSTR,RSTRCTRINSTR,
          TAKEINSTR,DROPINSTR,TRMINSTR,TRDINSTR,
          BRCODE,REMCODE,DIVCODE,DECODEINSTR,ENCODEINSTR;
  STRING  INFILE,OUTFILE;
  INTEGER ARRAY LNKREG,RANK,BETA,NINSTRS[1:50],DELTA,
                RHO[1:50,1:10],OPNDSTK[-3:200],
                OUTPRODLNK[1:20],TINITLIST[1:63],
                ORANK,OBETA,OOPNTYPE[1:20],ORHO,
                ODELTA[1:20,1:10],
                NSINSTRS,SCPTR,NRINSTRS,ACPTR,RCPTR,
                OPRIDENT[1:50],
                DATA[1:1000],FIXLIST[1:10];
BOOLEAN ARRAY   LADID,OPRTYPE[1:50],
                ACODE,OPTION[1:100],SCCODE,SOPTION,
                RCODE,ROPTION[1:50],
                SCODE[1:15,1:10,0:10],OPNTYPE[1:30];
  STRING ARRAY OPRNAME,OPNNAME[1:50];


  PROCEDURE SETSTK(STACK,LENGTH,FRAMESIZE);
        INTEGER LENGTH,FRAMESIZE; VALUE LENGTH,FRAMESIZE;
        INTEGER ARRAY STACK;
  BEGIN
    STACK[-3]:=0;
    STACK[-2]:=0;          !CURRENT LENGTH OF STACK;
    STACK[-1]:=LENGTH;     !MAXIMUM LENGTH OF STACK;
    STACK[0]:=FRAMESIZE;
  END;
```



```
PROCEDURE PUSH(STACK,ELEMENT); INTEGER ELEMENT;
   VALUE ELEMENT;   INTEGER ARRAY STACK;
BEGIN
  IF STACK[-2]+1  = STACK[-1] THEN
    BEGIN
      STACK[-2]:=STACK[-2]+1;
      STACK[STACK[-2]]:=ELEMENT;
    END
    ELSE
      BEGIN
        WRITE("STACK OVERFLOW -- POINTER AND LENGTH");
        NEWLINE; PRINT(STACK[-2]); PRINT(STACK[-1]);
      END
END;


PROCEDURE POP(STACK,ELEMENT); INTEGER ELEMENT;
             INTEGER ARRAY STACK;
BEGIN
  IF STACK[-2]   0 THEN
    BEGIN
      ELEMENT:=STACK[STACK[-2]];
      STACK[-2]:=STACK[-2]-1;
    END
    ELSE
      BEGIN
        WRITE("STACK UNDERFLOW -- POINTER AND LENGTH");
        NEWLINE; PRINT(STACK[-2]); PRINT(STACK[-1]);
      END
END;


PROCEDURE PUSHFRAME(STACK,FRAME);
      INTEGER ARRAY STACK,FRAME;

BEGIN INTEGER I;
  FOR I:= 1 UNTIL STACK[0] DO PUSH(STACK,FRAME[I]);
END;


PROCEDURE POPFRAME(STACK,FRAME);
      INTEGER ARRAY STACK,FRAME;

BEGIN INTEGER I;
  FOR I:=STACK[0] STEP -1 UNTIL 1 DO POP(STACK,FRAME[I]);
END;



INTEGER PROCEDURE INSTRADR(I);  INTEGER I;  VALUE I;
BEGIN
  INSTRADR:= I REM INSTRFIELD;
END;
```

```
INTEGER PROCEDURE SPLADR(I);  INTEGER I;  VALUE I;
BEGIN
  SPLADR:= (I DIV INSTRFIELD) REM SPLFIELD;
END;


INTEGER PROCEDURE LADR(I);  INTEGER I;  VALUE I;
BEGIN
  LADR:= (I DIV (INSTRFIELD*SPLFIELD)) REM LADRFIELD;
END;



PROCEDURE PRINTSTACK(STACK);
      INTEGER ARRAY STACK;
BEGIN       INTEGER ARRAY FRAME[1:STACK[0]];
            INTEGER I;
  IF STACK[-1]  = STACK[0] THEN
  BEGIN
    POPFRAME(STACK,FRAME);  PUSHFRAME(STACK,FRAME);
    WRITE("[2N]CURRENT TOP OF STACK  DEPTH OF STACK -- ");
    PRINT((STACK[-2] DIV STACK[0]),3);
    WRITE("[2N]LINK OUT (LADDER, SPLICE, INSTRUCTION)  ");
    PRINT(LADR(FRAME[1]),4); PRINT(SPLADR(FRAME[1]),4);
    PRINT(INSTRADR(FRAME[1]),4);
    WRITE("[N]DATA OUT (LADDER, SPLICE, INSTRUCTION)  ");
    PRINT(LADR(FRAME[2]),4); PRINT(SPLADR(FRAME[2]),4);
    PRINT(INSTRADR(FRAME[2]),4);
    WRITE("[N]LINK BACK  (LADDER, SPLICE, INSTRUCTION)" );
    PRINT(LADR(FRAME[3]),4); PRINT(SPLADR(FRAME[3]),4);
    PRINT(INSTRADR(FRAME[3]),4);
    WRITE("[N]TEMP OUT             "); PRINT(FRAME[4],4);
    WRITE("[N]LADR IN              "); PRINT(FRAME[5],4);
    WRITE("[N]TYPE OUT             "); PRINT(FRAME[6],4);
    WRITE("[N]RANK OUT             "); PRINT(FRAME[7],4);
    WRITE("[N]SHAPE OUT             ");
    FOR I:=1 UNTIL FRAME[7] DO PRINT(FRAME[7+I],4);
    NEWLINE(2);
  END
  ELSE WRITE("[2N]STACK DOESN'T HAVE FULL FRAME[2N]");
END;



PROCEDURE PRNTOPRS;
```



```
BEGIN  INTEGER I,J;
  NEWLINE(2);
  WRITE("THE NUMBER OF DEFINED OPERATORS IS ");
  PRINT(NOPRS,4);  NEWLINE(2);
  WRITE("SYMBOL TYPECODE  INSTRUCTION TEMPL AND MASK");
  FOR I:=1 UNTIL NOPRS DO
  BEGIN
    NEWLINE(2); SPACE(2);
    WRITE(OPRNAME[I]);  SPACE(4);  PRINTOCTAL(OPRTYPE[I]);
    FOR J:=1 UNTIL NINSTRS[I] DO
    BEGIN
      IF J#1 THEN
      BEGIN
        NEWLINE;
        SPACE(24);
      END
      ELSE SPACE(4);
      PRINTOCTAL(ACODE[ACPTR[I]-1+J]);  SPACE(4);
      PRINTOCTAL(OPTION[ACPTR[I]-1+J]);
    END;
  END;
  NEWLINE;
END;



PROCEDURE PRNTOPNS;

BEGIN  INTEGER I,J;
  NEWLINE(2);
  WRITE("THE NUMBER OF DEFINED OPERANDS IS ");
  PRINT(NOPNS,4);  NEWLINE(2);
  WRITE("SYMBOL    RANK          SHAPE");
  FOR I:=1 UNTIL NOPNS DO
  BEGIN
    NEWLINE;  SPACE(2);
    WRITE(OPNNAME[I]);  SPACE(4);  PRINT(RANK[I],4);
    SPACE(4);
    FOR J:=1 UNTIL RANK[I] DO PRINT(RHO[I,J],4);
  END;
  NEWLINE;
END;



PROCEDURE DEFOPRS;
BEGIN  INTEGER X,I,J;
  X:=INCHAN;
  INPUT(3,"DSK");  OPENFILE(3,"OPRDEF.DAT");
  SELECTINPUT(3);
```

```
    READ(NOPRS);
    FOR I:= 1 UNTIL NOPRS DO
    BEGIN
      OPRNAME[I]:=NEWSTRING(5,7);
      READ(OPRNAME[I]);
      IF OPRNAME[I]=" " THEN ASSGNOP:=I;
      READOCTAL(OPRTYPE[I]);
      READ(OPRIDENT[I]);
      READ(NINSTRS[I]);
      ACPTR[I]:=ACP;
      FOR J:=ACP UNTIL ACP-1+NINSTRS[I] DO
      BEGIN
        READOCTAL(ACODE[J]);  READOCTAL(OPTION[J]);
      END;
      ACP:=ACP+NINSTRS[I];
      IF NOCOMMUTEMASK AND OPRTYPE[I] THEN
      BEGIN
        RCPTR[I]:=RCP;
        READ(NRINSTRS[I]);
        FOR J:=RCP UNTIL RCP-1+NRINSTRS[I] DO
        BEGIN
          READOCTAL(RCODE[J]); READOCTAL(ROPTION[J]);
        END;
        RCP:=RCP+NRINSTRS[I];
        SCPTR[I]:=SCP;
        READ(NSINSTRS[I]);
        FOR J:=SCP UNTIL SCP-1+NSINSTRS[I] DO
        BEGIN
          READOCTAL(SCCODE[J]); READOCTAL(SOPTION[J]);
        END;
        SCP:=SCP+NSINSTRS[I];
      END
      ELSE
      BEGIN
        NRINSTRS[I]:=0;  NSINSTRS[I]:=0;
      END;
    END;
    CLOSEFILE(3);
    SELECTINPUT(X);
END;



PROCEDURE DEFOPNS;

BEGIN  INTEGER X,I,J;
    X:=INCHAN;
    INPUT(3,"DSK");  OPENFILE(3,"OPNDEF.DAT");
    SELECTINPUT(3);
    READ(NOPNS);
    FOR I:= 1 UNTIL NOPNS DO
    BEGIN
```



```
      READ(OPNNAME[I]);  READ(OPNTYPE[I]);
      READ(BETA[I]);     READ(RANK[I]);
      FOR J:=1 UNTIL RANK[I] DO
      BEGIN
        READ(RHO[I,J]); READ(DELTA[I,J]);
      END;
    END;
    CLOSEFILE(3);
    SELECTINPUT(X);
END;



PROCEDURE GETDAT;
BEGIN   INTEGER I,X;
    X:=INCHAN;
    INPUT(3,"DSK");  OPENFILE(3,"MEM.DAT");
    SELECTINPUT(3);
    READ(NWORDS);
    FOR I:=1 UNTIL NWORDS DO READ(DATA[I]);
    CLOSEFILE(3);
    SELECTINPUT(X);
END;



PROCEDURE SETUP;

BEGIN   INTEGER I,DOPSHFT,SOPSHFT;
    SETSTK(OPNDSTK,180,15);
    NAFB:=8;                        FLDSIZE:=2**NAFB;
    DOPSHFT:=FLDSIZE*FLDSIZE;
    SOPSHFT:=FLDSIZE;


    MOVCODE:=BOOL(27*DOPSHFT);
    CMPCODE:=BOOL(28*DOPSHFT);
    ADDCODE:=BOOL(13*DOPSHFT);
    DIVCODE:=BOOL(17*DOPSHFT);
    MULCODE:=BOOL(14*DOPSHFT);
    REMCODE:=BOOL(19*DOPSHFT);

    INCCODE:=BOOL(1*SOPSHFT) OR %40000;
    CLRCODE:=BOOL(4*SOPSHFT) OR %40000;
    TSTCODE:=BOOL(3*SOPSHFT) OR %40000;

    HALTCODE:=%0;
    BRCODE:=%100000;
    BNECODE:=%120000;
    BEQCODE:=%130000;
```

```
    ATPISOURCE:=%0;
    BINTYPE:=%1;          INTTYPE:=%1;          CRJCODE:=%100;
    MAXSINSTRS:=10;       OPRMASK:=%100;        OPNMASK:=%200;
    SYMBMASK:=%77;        SOURCETR:=%1;         SOURCETL:=%2;
    DESTTR:=%40;          SOURCEATPI:=%100;     DESTATPI:=%200;
    DESTT2:=%400;         SOURCETIN:=%4000;     DESTT1:=%10000;
    INSTRFIELD:=64;       SPLFIELD:=64;
    LADRFIELD:=64;        DESTTL:=%4;    ALLOCTEMPOUT:=%1000;
    DYADSCALMASK:=%100;

    MONADSCALMASK:=%200;
    DESTNEWTEMP:=%20;     SOURCENEWTEMP:=%10;
    OUTPRODINSTR:=%1;     REDUCINSTR:=%2;       MAXTEMPS:=31;
    ATPIDEST:=%0;         PISOURCE:=%400;       PIDEST:=%1;
    IPINSTR:=%3;          MAXLADRS:=10;         MAXOPNS:=30;
    MAXWORDS:=1000;       MAXRANK:=6;
    SCANINSTR:=%4;     MBRSHPINSTR:=%5;    INDXGENINSTR:=%6;
    INDXOFINSTR:=%7;   REVERSEINSTR:=%10;  RSTRCTRINSTR:=%12;
    SHAPEINSTR:=%11;      TAKEINSTR:=%13;     DROPINSTR:=%14;
    TRMINSTR:=%15;        TRDINSTR:=%16;
    EXITMARK:=BOOL(-1);
    RAVELCODE:=%17;       CATENATECODE:=%20;
    COMPRESSCODE:=%21;
    EXPANDCODE:=%22;      ENDMARK:=BOOL(-1);
    BETASOURCE:=%4300;
    DECODEINSTR:=%23;
    ENCODEINSTR:=%24;
    ILP:=1;               FLP:=0;
    ACP:=1;               RCP:=1;                 SCP:=1;
    BETADEST:=%3;         LMASK:=%400000000000;
    NOCOMMUTEMASK:=%400;
    FOR I:=1 UNTIL MAXLADRS DO OUTPRODLNK[I]:=0;
    FOR I:=1 UNTIL 2*MAXTEMPS+1 DO TINITLIST[I]:=0;

    DEFOPRS;
    DEFOPNS;
    GETDAT;
    NXTTMP:=32; NXTLAD:=1;  NXTLREG:=1;
    INPUT(3,"DSK"); OPENFILE(3,"C.DAT");  SELECTINPUT(3);
    READ(INFILE); READ(OUTFILE);
    CLOSEFILE(3);
  END;



COMMENT THIS BOOLEAN PROCEDURE CHECKS TO SEE IF TWO BOOLEAN
VARIABLES ARE EXACTLY EQUAL;

  BOOLEAN PROCEDURE MATCH(A,B); BOOLEAN A,B; VALUE A,B;
  BEGIN
```



```
    MATCH:=INT(A EQV B) = -1;
  END;


  INTEGER PROCEDURE SMATCH(S,SA,SAL);  STRING S;
     STRING ARRAY SA; INTEGER SAL;  VALUE S,SAL;

  BEGIN  INTEGER I,J,X;
    X:=0;
    FOR I:= 1 UNTIL SAL DO IF S=SA[I] THEN
    BEGIN
      X:=I;  I:=SAL;
    END;
    SMATCH:=X;
  END;



  BOOLEAN PROCEDURE TF(X);  BOOLEAN X; VALUE X;
  BEGIN
    TF:= IF X THEN TRUE ELSE FALSE;
  END;



  STRING PROCEDURE INTSTRNG(I);
        INTEGER I;       VALUE I;
  BEGIN
        INTEGER L,X;     REAL Y;    STRING A;

    A:=NEWSTRING(10,7);
    X:=OUTCHAN;              Y:=I;
    OUTPUT(20,A);            SELECTOUTPUT(20);
    L:=ENTIER(LN(ABS(Y))/LN(10.));
    PRINT(I,L+1);
    SELECTOUTPUT(X);
    INTSTRNG:=A;
  END;



  STRING PROCEDURE MCONCAT(A,B);
        STRING A,B;      VALUE A,B;
  BEGIN
        INTEGER LA,LB,I;
        STRING  C;

    LA:=LENGTH(A);       LB:=LENGTH(B);
    C:=NEWSTRING(LA+LB,7);
    FOR I:=1 UNTIL LA DO C.[I]:=A.[I];
    FOR I:=1 UNTIL LB DO C.[LA+I]:=B.[I];
```

```
  MCONCAT:=C;
END;



INTEGER PROCEDURE STRNGINT(S);
      STRING S;
BEGIN
      INTEGER I,J,K,L;        STRING LS;

  LS:=NEWSTRING(5,7);
  LS:=COPY(S);
  L:=LENGTH(LS);       K:=0;
  FOR I:=1 UNTIL L DO
  BEGIN
    J:=LS.[I];
    IF J =$/0/ AND J =$/9/ THEN K:=K*10+J-$/0/
    ELSE WRITE("[N]INPUT NOT A DIGIT");
  END;
  STRNGINT:=K;
END;



BOOLEAN PROCEDURE EOF(CHAN);
      INTEGER CHAN;
BEGIN
  EOF:=TF(IOCHAN(CHAN) AND %100);
END;



STRING PROCEDURE GNXTLEX(ENDFILE,CHAN);  BOOLEAN ENDFILE;
      INTEGER CHAN;   VALUE CHAN;

BEGIN  INTEGER S,I;  STRING ST;
  ST:=NEWSTRING(10,7);
  S:= $/ /;
  WHILE S  = $/ / AND NOT(EOF(CHAN)) DO INSYMBOL(S);
  IF NOT(EOF(CHAN)) THEN
  BEGIN
    FOR I:=1,I+1 WHILE S   $/ / AND NOT(EOF(CHAN)) DO
    BEGIN
```



```
      ST.[I]:=S;
      INSYMBOL(S);
    END;
    GNXTLEX:=COPY(ST,I-1);
  END
  ELSE GNXTLEX:="";
  ENDFILE:=EOF(CHAN);
  DELETE(ST);
END;



BOOLEAN PROCEDURE NEXTSYMB(SYMB,CHAN);  BOOLEAN SYMB;
      INTEGER CHAN; VALUE CHAN;

BEGIN  STRING S;  BOOLEAN EOF;
  EOF:=FALSE;  SYMB:=%0;
  IF NOT EOF AND LENGTH((S:=GNXTLEX(EOF,CHAN)))#0 THEN
  BEGIN
    OPRNUM:=SMATCH(S,OPRNAME,NOPRS);
    OPNNUM:=SMATCH(S,OPNNAME,NOPNS);
    SYMB:=((BOOL(OPRNUM) OR OPRMASK) AND (OPRNUM # 0)) OR
          ((BOOL(OPNNUM) OR OPNMASK) AND (OPNNUM # 0));
    IF MATCH (NOT SYMB, TRUE) THEN
    BEGIN
      WRITE("UNDEFINED SYMBOL "); WRITE(S); NEWLINE;
      NEXTSYMB(SYMB,CHAN);
    END;
  END;
  NEXTSYMB:=NOT EOF;
END;



BOOLEAN PROCEDURE SCALAR(FRAME);
      INTEGER ARRAY FRAME;
BEGIN
      INTEGER I,K,PROD;

  PROD:=1;
  IF BOOL(FRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(FRAME[1]) AND SYMBMASK);
    IF RANK[K]  = 0 THEN
    BEGIN
      FOR I:=1 UNTIL RANK[K] DO
        PROD:=IF RHO[K,I] =0 THEN PROD*RHO[K,I] ELSE 0;
    END
    ELSE PROD:=-1;
  END
  ELSE
```

```
  BEGIN
    IF FRAME[7] =0 THEN
    BEGIN
      FOR I:=1 UNTIL FRAME[7] DO
        PROD:=IF FRAME[I+7] =0 THEN PROD*FRAME[I+7] ELSE 0;
    END
    ELSE PROD:=-1;
  END;
  SCALAR:=PROD=1;
END;




BOOLEAN PROCEDURE VECTOR(FRAME);
      INTEGER ARRAY FRAME;
BEGIN

  VECTOR:=1 =(IF BOOL(FRAME[1]) AND LMASK THEN
      RANK[INT(BOOL(FRAME[1]) AND SYMBMASK)]
      ELSE FRAME[7]);
END;




PROCEDURE GENSCODE(LN,RANK);
      INTEGER LN,RANK; VALUE LN,RANK;

BEGIN INTEGER I;
  FOR I:=1 UNTIL 2*RANK+1 DO
  BEGIN
    SCODE[LN,I,0]:=%0;
  END;
END;


PROCEDURE ASSIGNTEMP(T); INTEGER T;
BEGIN
  T:=NXTTMP; NXTTMP:=NXTTMP+1;
END;

PROCEDURE ASSIGNLNUM(T,OPNNUM); INTEGER T,OPNNUM; VALUE OPNNUM;
BEGIN  INTEGER I;
  T:=NXTLAD; NXTLAD:=NXTLAD+1;
  IF OPNNUM#0 THEN
  BEGIN
    LADID[T]:=BOOL(OPNNUM);
```



```
    ORANK[T]:=RANK[OPNNUM];  OBETA[T]:=BETA[OPNNUM];
    OOPNTYPE[T]:=INT(OPNTYPE[OPNNUM]);
    FOR I:=1 UNTIL RANK[OPNNUM] DO
    BEGIN
      ORHO[T,I]:=RHO[OPNNUM,I];
      ODELTA[T,I]:=DELTA[OPNNUM,I];
    END;
  END;
END;

PROCEDURE ASSIGNLREG(T); INTEGER T;
BEGIN
  T:=NXTLREG; NXTLREG:=NXTLREG+1;
END;


INTEGER PROCEDURE MAKESPLADR(LADR,SPLICE,INSTR);
    INTEGER LADR,SPLICE,INSTR;  VALUE LADR,SPLICE,INSTR;
BEGIN
  MAKESPLADR:=(LADR*SPLFIELD+SPLICE)*INSTRFIELD+INSTR;
END;


PROCEDURE MAKERCODE(CODE,LOPNFRAME,ROPNFRAME,OP,NEWTEMP,T2);
      INTEGER OP,NEWTEMP,T2;      VALUE OP,NEWTEMP,T2;
INTEGER ARRAY ROPNFRAME,LOPNFRAME;  BOOLEAN ARRAY CODE;

BEGIN  INTEGER N,I;     BOOLEAN FILWRD;
  N:=NRINSTRS[OP];
  FOR I:=1 UNTIL N DO
  BEGIN
    CODE[I]:=RCODE[RCPTR[OP]-1+I];
    FILWRD:=ROPTION[RCPTR[OP]-1+I];
    IF FILWRD AND SOURCETR THEN
        CODE[I]:=CODE[I] OR BOOL(ROPNFRAME[4]*FLDSIZE);
    IF FILWRD AND SOURCETL THEN
        CODE[I]:=CODE[I] OR BOOL(LOPNFRAME[4]*FLDSIZE);
    IF FILWRD AND DESTTL THEN
        CODE[I]:=CODE[I] OR BOOL(LOPNFRAME[4]);
    IF FILWRD AND SOURCENEWTEMP THEN
        CODE[I]:=CODE[I] OR BOOL(NEWTEMP*FLDSIZE);
    IF FILWRD AND DESTNEWTEMP THEN
        CODE[I]:=CODE[I] OR BOOL(NEWTEMP);
    IF FILWRD AND DESTTR THEN
        CODE[I]:=CODE[I] OR BOOL(ROPNFRAME[4]);
    IF FILWRD AND SOURCEATPI THEN
        CODE[I]:=CODE[I] OR ATPISOURCE;
    IF FILWRD AND DESTATPI THEN
        CODE[I]:=CODE[I] OR ATPIDEST;
    IF FILWRD AND DESTT2 THEN
        CODE[I]:=CODE[I] OR BOOL(T2);
  END;
END;
```

```
PROCEDURE MAKESCODE(CODE,OPNFRAME,OP,T1,T2);
      INTEGER OP,T1,T2;     VALUE OP,T1,T2;
INTEGER ARRAY OPNFRAME;     BOOLEAN ARRAY CODE;

BEGIN  INTEGER N,I;     BOOLEAN FILWRD;
  N:=NSINSTRS[OP];
  FOR I:=1 UNTIL N DO
  BEGIN
    CODE[I]:=SCCODE[SCPTR[OP]-1+I];
    FILWRD:=SOPTION[SCPTR[OP]-1+I];
    IF FILWRD AND SOURCETIN THEN
        CODE[I]:=CODE[I] OR BOOL(OPNFRAME[4]*FLDSIZE);
    IF FILWRD AND DESTT1 THEN CODE[I]:=CODE[I] OR BOOL(T1);
    IF FILWRD AND DESTT2 THEN CODE[I]:=CODE[I] OR BOOL(T2);
  END;
END;


PROCEDURE MAKECODE(CODE,LOPNFRAME,ROPNFRAME,OP,NEWTEMP);
      INTEGER OP,NEWTEMP;   VALUE OP,NEWTEMP;
INTEGER ARRAY ROPNFRAME,LOPNFRAME;  BOOLEAN ARRAY CODE;

BEGIN  INTEGER N,I;     BOOLEAN FILWRD;
  N:=NINSTRS[OP];
  FOR I:=1 UNTIL N DO
  BEGIN
    CODE[I]:=ACODE[ACPTR[OP]-1+I];
    FILWRD:=OPTION[ACPTR[OP]-1+I];
    IF FILWRD AND SOURCETR THEN
        CODE[I]:=CODE[I] OR BOOL(ROPNFRAME[4]*FLDSIZE);
    IF FILWRD AND SOURCETL THEN
        CODE[I]:=CODE[I] OR BOOL(LOPNFRAME[4]*FLDSIZE);
    IF FILWRD AND DESTTL THEN
        CODE[I]:=CODE[I] OR BOOL(LOPNFRAME[4]);
    IF FILWRD AND SOURCENEWTEMP THEN
        CODE[I]:=CODE[I] OR BOOL(NEWTEMP*FLDSIZE);
    IF FILWRD AND DESTNEWTEMP THEN
        CODE[I]:=CODE[I] OR BOOL(NEWTEMP);
    IF FILWRD AND DESTTR THEN
        CODE[I]:=CODE[I] OR BOOL(ROPNFRAME[4]);
    IF FILWRD AND SOURCEATPI THEN
        CODE[I]:=CODE[I] OR ATPISOURCE;
    IF FILWRD AND DESTATPI THEN
        CODE[I]:=CODE[I] OR ATPIDEST;
  END;
END;
```



```
PROCEDURE INSERTCODE(CODE,WHERE,N); INTEGER WHERE,N;
   VALUE WHERE,N;  BOOLEAN ARRAY CODE;

BEGIN  INTEGER I,J,K,L;
  I:=LADR(WHERE);
  J:=SPLADR(WHERE);
  K:=INSTRADR(WHERE);
  IF N+INT(SCODE[I,J,0])  = MAXSINSTRS  THEN
  BEGIN
    FOR L:=INT(SCODE[I,J,0]) STEP -1 UNTIL K DO
        SCODE[I,J,L+N]:=SCODE[I,J,L];
    FOR L:=1 UNTIL N DO SCODE[I,J,K+L-1]:=CODE[L];
    SCODE[I,J,0]:=BOOL(INT(SCODE[I,J,0])+N);
  END
  ELSE
  BEGIN   NEWLINE;
    WRITE("TOO MUCH SPLICE CODE"); PRINT(N,4);
    PRINT(WHERE,6);
  END;
END;


PROCEDURE UPDATEPOINTER(PTRARRAY,START,STOP,WHERE,DELTA);
 INTEGER ARRAY PTRARRAY;  INTEGER START,STOP,WHERE,DELTA;
                          VALUE   START,STOP,WHERE,DELTA;

BEGIN  INTEGER I,J,K,L;
  I:=LADR(WHERE);
  J:=SPLADR(WHERE);
  K:=INSTRADR(WHERE);
  FOR L:= START UNTIL STOP DO
    IF LADR(PTRARRAY[L]) = I AND
       SPLADR(PTRARRAY[L]) = J AND
       INSTRADR(PTRARRAY[L])  = K THEN
         PTRARRAY[L]:=
              MAKESPLADR(I,J,INSTRADR(PTRARRAY[L])+DELTA);
END;


BOOLEAN PROCEDURE CONFORMCHK(LOPNFRAME,ROPNFRAME,OP);
INTEGER ARRAY LOPNFRAME,ROPNFRAME;
INTEGER OP;  VALUE OP;

BEGIN  INTEGER PRODL,PRODR,RRNK,LRNK,I;  BOOLEAN OK;
  OK:=TRUE;
  RRNK:=ROPNFRAME[7];  LRNK:=LOPNFRAME[7];
  PRODL:=1;   PRODR:=1;
  FOR I:=1 UNTIL LRNK DO PRODL:=PRODL*LOPNFRAME[7+I];
  FOR I:=1 UNTIL RRNK DO PRODR:=PRODR*ROPNFRAME[7+I];
  IF NOT(PRODL=0 OR PRODL=1 OR PRODR=0 OR PRODR=1) THEN
```

```
  BEGIN
    IF LRNK=RRNK THEN
    BEGIN
      FOR I:=1 UNTIL LRNK DO
      IF LOPNFRAME[I+7]#ROPNFRAME[I+7] THEN OK:=FALSE;
    END
    ELSE OK:=FALSE;
  END;
  CONFORMCHK:=OK;
END;



BOOLEAN PROCEDURE IPCONFORMCHK(LOPN,ROPN,LOPR,ROPR);
    INTEGER LOPN,ROPN,LOPR,ROPR;
    VALUE LOPN,ROPN,LOPR,ROPR;
BEGIN   INTEGER RRNK,LRNK,PRODR,PRODL,I;
  IF OPRTYPE[LOPR] AND OPRTYPE[ROPR]
                      AND DYADSCALMASK THEN
  BEGIN
    RRNK:=RANK[ROPN];  LRNK:=RANK[LOPN];
    PRODR:=1;          PRODL:=1;
    FOR I:=1 UNTIL RRNK DO PRODR:=PRODR*RHO[ROPN,I];
    FOR I:=1 UNTIL LRNK DO PRODL:=PRODL*RHO[LOPN,I];
    IPCONFORMCHK:= PRODL=0 OR PRODL=1 OR PRODR=0 OR
                   PRODR=1 OR RHO[LOPN,LRNK]=RHO[ROPN,1];
  END
  ELSE IPCONFORMCHK:=FALSE;
END;


PROCEDURE COMPDELTA(DELTA,RHO,G,RANK);
  INTEGER RANK; INTEGER ARRAY G,RHO,DELTA;
  VALUE RANK,G,RHO;
BEGIN   INTEGER I,J;
  DELTA[RANK]:=G[RANK];
  FOR I:=RANK-1 STEP -1 UNTIL 1 DO
    DELTA[I]:=G[I]+DELTA[I+1]-RHO[I+1]*G[I+1];
END;



PROCEDURE COMPG(G,RHO,DELTA,RANK);
  INTEGER RANK; INTEGER ARRAY G,RHO,DELTA;
  VALUE RANK,DELTA,RHO;
BEGIN   INTEGER I,J;
  G[RANK]:=DELTA[RANK];
  FOR I:=RANK-1 STEP -1 UNTIL 1 DO
    G[I]:=DELTA[I]+RHO[I+1]*G[I+1]-DELTA[I+1];
END;
```



```
PROCEDURE LTRANSPOSE(L,TVEC);
    INTEGER L; INTEGER ARRAY TVEC;  VALUE L,TVEC;
BEGIN     INTEGER RANK,MAX,I,MIN,J;
          INTEGER ARRAY RHOIN,RHOOUT,GIN,GOUT,
                        DELTA[1:ORANK[L]];
  RANK:=ORANK[L];
  FOR I:=1 UNTIL RANK DO
  BEGIN
    RHOIN[I]:=ORHO[L,I];  DELTA[I]:=ODELTA[L,I];
  END;
  COMPG(GIN,RHOIN,DELTA,RANK);
  MAX:=TVEC[1];
  FOR I:=2 UNTIL RANK DO MAX:=IMAX(MAX,TVEC[I]);
  ORANK[L]:=MAX;
  FOR I:=1 UNTIL MAX DO
  BEGIN
    MIN:=(2**34-1)+2**34;        GOUT[I]:=0;
    FOR J:=1 UNTIL RANK DO
    IF I=TVEC[J] THEN
    BEGIN
      MIN:=IMIN(MIN,RHOIN[J]);  GOUT[I]:=GOUT[I]+GIN[J];
    END;
    RHOOUT[I]:=MIN;
  END;
  COMPDELTA(DELTA,RHOOUT,GOUT,MAX);
  FOR I:=1 UNTIL MAX DO
  BEGIN
    ORHO[L,I]:=RHOOUT[I]; ODELTA[L,I]:=DELTA[I];
  END;
END;


INTEGER PROCEDURE LSPLIT(L,N);
      INTEGER L,N;  VALUE L,N;
BEGIN   INTEGER I,L2,DELTAADJ;
  IF N 0 AND N ORANK[L] THEN
  BEGIN
    ASSIGNLNUM(L2,0);
    ORANK[L2]:=ORANK[L]-N;
    OBETA[L2]:=0;      OOPNTYPE[L2]:=OOPNTYPE[L];
    FOR I:=1 UNTIL ORANK[L2] DO
    BEGIN
      ORHO[L2,I]:=ORHO[L,I+N];
      ODELTA[L2,I]:=ODELTA[L,I+N];
    END;
    DELTAADJ:=0;
    FOR I:= ORANK[L] STEP -1 UNTIL N+1 DO
    BEGIN
      DELTAADJ:=DELTAADJ+ODELTA[L,I];
      DELTAADJ:=DELTAADJ*ORHO[L,I]-ODELTA[L,I];
    END;
```

```
    ODELTA[L,N]:=ODELTA[L,N]+DELTAADJ;
    ORANK[L]:=N;
    LSPLIT:=L2;
  END
  ELSE LSPLIT:=0;
END;



PROCEDURE MERGEOPFRAME
              (MFRAME,LOPNFRAME,ROPNFRAME,TEMPOUT);
      INTEGER ARRAY MFRAME,LOPNFRAME,ROPNFRAME;
      INTEGER TEMPOUT; VALUE TEMPOUT;

BEGIN  INTEGER I;
  MFRAME[1]:=ROPNFRAME[1];
  MFRAME[2]:=MFRAME[1];
  MFRAME[3]:=MFRAME[1];
  MFRAME[4]:=TEMPOUT;
  MFRAME[5]:=LOPNFRAME[5];
  MFRAME[6]:=LOPNFRAME[6];
  MFRAME[7]:=ROPNFRAME[7]+LOPNFRAME[7];
  FOR I:=1 UNTIL LOPNFRAME[7] DO
      MFRAME[I+7]:=LOPNFRAME[I+7];
  FOR I:=1 UNTIL ROPNFRAME[7] DO
      MFRAME[I+7+LOPNFRAME[7]]:=ROPNFRAME[I+7];
END;



PROCEDURE MERGEFRAME
      (MFRAME,LOPNFRAME,ROPNFRAME,TEMPOUT);
      INTEGER ARRAY MFRAME,LOPNFRAME,ROPNFRAME;
      INTEGER TEMPOUT; VALUE TEMPOUT;

BEGIN  INTEGER I;
  MFRAME[1]:=LOPNFRAME[2];
  MFRAME[2]:=MFRAME[1];
  MFRAME[3]:=ROPNFRAME[3];
  MFRAME[4]:=TEMPOUT;
  MFRAME[5]:=ROPNFRAME[5];
  FOR I:=6 UNTIL 7+LOPNFRAME[7] DO
      MFRAME[I]:=LOPNFRAME[I];
END;



INTEGER PROCEDURE DEFLFRAME(FRAME);
      INTEGER ARRAY FRAME;
```



```
BEGIN
      INTEGER L,K,I,RNK;
  K:=INT(BOOL(FRAME[1]) AND SYMBMASK);
  ASSIGNLNUM(L,K);
  RNK:=RANK[K];
  GENSCODE(L,RNK);
  FRAME[1]:=FRAME[2]:=FRAME[3]:=MAKESPLADR(L,RNK+1,1);
  FRAME[4]:=0;
  FRAME[5]:=L;
  FRAME[6]:=INT(OPNTYPE[K]);
  FRAME[7]:=RNK;
  FOR I:=1 UNTIL RNK DO FRAME[I+7]:=RNO[K,I];
  DEFLFRAME:=L;
END;



INTEGER PROCEDURE FNXTSPL(WHERE);
      INTEGER WHERE;  VALUE WHERE;
BEGIN
      INTEGER L,S,RNK,J;
  L:=LADR(WHERE);      S:=SPLADR(WHERE);
  RNK:=ORANK[L];
  J:=S+1;
  IF J 2*RNK+1 THEN
  BEGIN
    IF OUTPRODLNK[L]#0 THEN
    BEGIN
      L:=OUTPRODLNK[L];
      WHERE:=MAKESPLADR(L,ORANK[L]+1,1);
      WHERE:=FNXTSPL(WHERE);
    END
    ELSE
    BEGIN
      WHERE:=0;  WRITE("[N]SPLICE ADVANCE FAILURE");
    END;
  END
  ELSE WHERE:=MAKESPLADR(L,J,1);
  FNXTSPL:=WHERE;
END;



INTEGER PROCEDURE BNXTSPL(WHERE);
      INTEGER WHERE;  VALUE WHERE;
BEGIN
      INTEGER L,S,RNK,J;
  L:=LADR(WHERE);      S:=SPLADR(WHERE);
  RNK:=ORANK[L];
  J:=2*RNK-S+1;
```

```
  IF J=0 THEN
  BEGIN
    IF OUTPRODLNK[L]#0 THEN
    BEGIN
      L:=OUTPRODLNK[L];
      WHERE:=MAKESPLADR(L,ORANK[L]+1,1);
      WHERE:=BNXTSPL(WHERE);
    END
    ELSE
    BEGIN
      WHERE:=0;  WRITE("[N]SPLICE BACKTRACK FAILURE");
    END;
  END
  ELSE WHERE:=MAKESPLADR(L,J,1);
  BNXTSPL:=WHERE;
END;
```

```
INTEGER PROCEDURE MAKEFIXLIST(LADR,ITEM,TEMP);
      INTEGER LADR,ITEM,TEMP;  VALUE LADR,ITEM,TEMP;
BEGIN
  MAKEFIXLIST:=4096*LADR + 64*ITEM +TEMP;
END;
```

```
PROCEDURE LDROP(L,DROPVEC);
      INTEGER L;     VALUE L;     INTEGER ARRAY DROPVEC;
BEGIN
      INTEGER I,RANK,SUM;
      INTEGER ARRAY DELTA,G,RHO[1:ORANK[L]];

  RANK:=ORANK[L];
  FOR I:= 1 UNTIL RANK DO
  BEGIN
    RHO[I]:=ORHO[L,I];     DELTA[I]:=ODELTA[L,I];
  END;
  COMPG(G,RHO,DELTA,RANK);
  SUM:=0;
  FOR I:=1 UNTIL RANK DO
  BEGIN
    IF DROPVEC[I]   0 THEN SUM:=SUM+DROPVEC[I]*G[I];
    RHO[I]:=RHO[I]-ABS(DROPVEC[I]);
  END;
  OBETA[L]:=OBETA[L]+SUM;
```



```
  COMPDELTA(DELTA,RHO,G,RANK);
  FOR I:=1 UNTIL RANK DO
  BEGIN
    ORHO[L,I]:=RHO[I];          ODELTA[L,I]:=DELTA[I];
  END;
END;
```

```
PROCEDURE LTAKE(L,TAKEVEC);
      INTEGER L;     VALUE L;     INTEGER ARRAY TAKEVEC;
BEGIN
      INTEGER I,RANK,SUM;
      INTEGER ARRAY DELTA,G,RHO[1:ORANK[L]];

  RANK:=ORANK[L];
  FOR I:= 1 UNTIL RANK DO
  BEGIN
    RHO[I]:=ORHO[L,I];     DELTA[I]:=ODELTA[L,I];
  END;
  COMPG(G,RHO,DELTA,RANK);
  SUM:=0;
  FOR I:=1 UNTIL RANK DO
  BEGIN
    IF TAKEVEC[I]   0 THEN
              SUM:=SUM+(RHO[I]+TAKEVEC[I])*G[I];
    RHO[I]:=ABS(TAKEVEC[I]);
  END;
  OBETA[L]:=OBETA[L]+SUM;
  COMPDELTA(DELTA,RHO,G,RANK);
  FOR I:=1 UNTIL RANK DO
  BEGIN
    ORHO[L,I]:=RHO[I];          ODELTA[L,I]:=DELTA[I];
  END;
END;
```

```
PROCEDURE LREVERSE(L,SUB);
      INTEGER L,SUB;  VALUE L,SUB;
BEGIN
      INTEGER RNK,I;
      INTEGER ARRAY RHO,G,DELTA[1:ORANK[L]];

  RNK:=ORANK[L];
  FOR I:=1 UNTIL RNK DO
  BEGIN
```

```
    RHO[I]:=ORHO[L,I];  DELTA[I]:=ODELTA[L,I];
  END;
  COMPG(G,RHO,DELTA,RNK);
  G[SUB]:=-G[SUB];
  OBETA[L]:=OBETA[L]-G[SUB]*(RHO[SUB]-1);
  COMPDELTA(DELTA,RHO,G,RNK);
  FOR I:=1 UNTIL RNK DO ODELTA[L,I]:=DELTA[I];
END;



PROCEDURE REVERSE(STACK);
      INTEGER ARRAY STACK;
BEGIN      INTEGER K,PRODL,I,X,L,RNK,T1,WHERE;
           INTEGER ARRAY LOPNFRAME,ROPNFRAME[1:STACK[0]];
           BOOLEAN ARRAY CODE[1:10];

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    PRODL:=1;
    FOR I:=1 UNTIL RANK[K] DO PRODL:=PRODL*RHO[K,I];
    IF PRODL#1 THEN
      WRITE("[N]LEFT ARG TO .RV NOT SINGLE ELEMENT")
    ELSE
    BEGIN
      X:=DATA[BETA[K]];
      IF BOOL(ROPNFRAME[1]) AND LMASK THEN
      BEGIN
        K:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
        ASSIGNLNUM(L,K);
        RNK:=ORANK[L];
        GENSCODE(L,RNK);
        IF X 0 AND X =RANK[K] THEN LREVERSE(L,X);
        ASSIGNTEMP(T1);
        WHERE:=MAKESPLADR(L,RNK+1,1);
        CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T1);
        INSERTCODE(CODE,WHERE,1);
        ROPNFRAME[3]:=ROPNFRAME[2]:=ROPNFRAME[1]:=WHERE+1;
        ROPNFRAME[4]:=T1;
        ROPNFRAME[5]:=L;
        ROPNFRAME[6]:=INT(OPNTYPE[K]);
        ROPNFRAME[7]:=RNK;
        FOR I:=1 UNTIL RNK DO ROPNFRAME[I+7]:=RHO[K,I];
        PUSHFRAME(STACK,ROPNFRAME);
      END
      ELSE
        WRITE("[N]RT ARG OF .RV MUST BE SIMPLE LADDER");
    END;
```



```
  END
  ELSE WRITE("[N]LEFT ARG OF .RV MUST BE SIMPLE LADDER");
END;



PROCEDURE DECODE(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY MFRAME,LOPNFRAME,
                    ROPNFRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER RLENGTH,LLENGTH,I,L,R,TR,K,T,LR;
      BOOLEAN RSCALAR,LSCALAR,RVECTOR,LVECTOR;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  RVECTOR:=VECTOR(ROPNFRAME);
  LVECTOR:=VECTOR(LOPNFRAME);
  RSCALAR:=SCALAR(ROPNFRAME);
  LSCALAR:=SCALAR(LOPNFRAME);
  RLENGTH:=IF BOOL(ROPNFRAME[1]) AND LMASK THEN
              RHO[INT(BOOL(ROPNFRAME[1]) AND SYMBMASK),1]
            ELSE ROPNFRAME[8];
  LLENGTH:=IF BOOL(LOPNFRAME[1]) AND LMASK THEN
              RHO[INT(BOOL(LOPNFRAME[1]) AND SYMBMASK),1]
            ELSE LOPNFRAME[8];
  IF LSCALAR AND (RSCALAR OR RVECTOR AND ROPNFRAME[8] =0)
      OR LVECTOR AND (RSCALAR OR RVECTOR AND
      ROPNFRAME[8] =0
      AND RLENGTH=LLENGTH) THEN
  BEGIN
    IF BOOL(ROPNFRAME[1]) AND LMASK THEN
    BEGIN
      R:=DEFLFRAME(ROPNFRAME);
      ASSIGNTEMP(TR);
      CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(TR);
      INSERTCODE(CODE,ROPNFRAME[2],1);
      UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],1);
      ROPNFRAME[4]:=TR;
    END;
    IF LSCALAR THEN
    BEGIN
      IF BOOL(LOPNFRAME[1]) AND LMASK THEN
      BEGIN
        K:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
        ASSIGNLNUM(L,0);
        ORANK[L]:=1;        OOPNTYPE[L]:=INT(OPNTYPE[K]);
        OBETA[L]:=BETA[K];     ODELTA[L,1]:=0;
        ORHO[L,1]:=ROPNFRAME[8];
```

```
      GENSCODE(L,1);
      ASSIGNTEMP(T);          LOPNFRAME[4]:=T;
      CODE[1]:=MOVCODE OR BETASOURCE OR PIDEST;
      CODE[2]:=MOVCODE OR ATPISOURCE OR BOOL(T);
      I:=MAKESPLADR(L,1,1);
      INSERTCODE(CODE,I,2);
      LOPNFRAME[1]:=LOPNFRAME[2]:=LOPNFRAME[3]:=
                    MAKESPLADR(L,2,1);
      LOPNFRAME[5]:=L;        LOPNFRAME[6]:=OOPNTYPE[L];
      LOPNFRAME[7]:=1;        LOPNFRAME[8]:=ORHO[L,1];
    END
    ELSE
    BEGIN
      ASSIGNLNUM(L,0);
      ORANK[L]:=1;            OOPNTYPE[L]:=LOPNFRAME[6];
      OBETA[L]:=0;            ODELTA[L,1]:=0;
      ORHO[L,1]:=ROPNFRAME[8];
      GENSCODE(L,1);
      ASSIGNLREG(LR);         LNKREG[LR]:=LOPNFRAME[5];
      CODE[1]:=CRJCODE OR BOOL(LR);
      I:=MAKESPLADR(L,1,1);
      INSERTCODE(CODE,I,1);
      LOPNFRAME[1]:=LOPNFRAME[2]:=LOPNFRAME[3]:=
            MAKESPLADR(L,2,1);
      LOPNFRAME[5]:=L;
      LOPNFRAME[7]:=1;        LOPNFRAME[8]:=ORHO[L,1];
    END;
  END;
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
          L:=DEFLFRAME(LOPNFRAME);
  ASSIGNTEMP(T);
  CODE[1]:=CLRCODE OR BOOL(T);
  I:=BNXTSPL(LOPNFRAME[2]);
  INSERTCODE(CODE,I,1);
  ASSIGNLREG(LR);             LNKREG[LR]:=ROPNFRAME[5];
  CODE[1]:=MULCODE OR BOOL(FLDSIZE*LOPNFRAME[4])
          OR BOOL(T);
  CODE[2]:=CRJCODE OR BOOL(LR);
  CODE[3]:=ADDCODE OR BOOL(FLDSIZE*ROPNFRAME[4])
          OR BOOL(T);
  INSERTCODE(CODE,LOPNFRAME[2],3);
  CODE[1]:=CRJCODE OR BOOL(LR);
  INSERTCODE(CODE,ROPNFRAME[1],1);
  MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=
          FNXTSPL(LOPNFRAME[2]);
  MFRAME[4]:=T;
  MFRAME[5]:=LOPNFRAME[5];
  MFRAME[6]:=INT(INTTYPE);
  MFRAME[7]:=1;
  MFRAME[8]:=1;
  PUSHFRAME(STACK,MFRAME);
END
ELSE WRITE("[N]BAD ARG MIX TO DECODE");
```



```
END;



PROCEDURE ENCODE(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY MFRAME,LOPNFRAME,
                    ROPNFRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER T1,L,R,LR,I,T2,T3;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF SCALAR(ROPNFRAME) AND VECTOR(LOPNFRAME) THEN
  BEGIN
    IF BOOL(ROPNFRAME[1]) AND LMASK THEN
    BEGIN
      R:=DEFLFRAME(ROPNFRAME);
      ASSIGNTEMP(T1);
      CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T1);
      INSERTCODE(CODE,ROPNFRAME[2],1);
      UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],1);
      ROPNFRAME[4]:=T1;
    END;
    IF BOOL(LOPNFRAME[1]) AND LMASK THEN
            L:=DEFLFRAME(LOPNFRAME);
    ASSIGNLREG(LR);  LNKREG[LR]:=ROPNFRAME[5];
    CODE[1]:=CRJCODE OR BOOL(LR);
    INSERTCODE(CODE,ROPNFRAME[1],1);
    I:=BNXTSPL(LOPNFRAME[2]);
    INSERTCODE(CODE,I,1);
    ASSIGNTEMP(T2);  ASSIGNTEMP(T3);
    CODE[1]:=MOVCODE OR BOOL(FLDSIZE*ROPNFRAME[4])
                     OR BOOL(T3);
    CODE[2]:=MOVCODE OR BOOL(FLDSIZE*LOPNFRAME[4])
                     OR BOOL(T2);
    CODE[3]:=TSTCODE OR BOOL(T2);
    CODE[4]:=BNECODE OR %2;
    CODE[5]:=CLRCODE OR BOOL(ROPNFRAME[4]);
    CODE[6]:=BRCODE  OR %2;
    CODE[7]:=REMCODE OR BOOL(FLDSIZE*T2) OR BOOL(T3);
    CODE[8]:=DIVCODE OR BOOL(FLDSIZE*T2) OR
                     BOOL(ROPNFRAME[4]);
    INSERTCODE(CODE,LOPNFRAME[2],8);
    UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],8);
    MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=LOPNFRAME[2];
     MFRAME[4]:=T3;  MFRAME[5]:=LOPNFRAME[5];
    MFRAME[6]:=INT(INTTYPE);
    MFRAME[7]:=1;  MFRAME[8]:=LOPNFRAME[8];
```

```
    PUSHFRAME(STACK,MFRAME);
  END
  ELSE WRITE("[N]BAD ARG MIX TO ENCODE");
END;



PROCEDURE COMPRESS(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY MFRAME,LOPNFRAME,
                    ROPNFRAME[1:STACK[0]];
      BOOLEAN ARRAY  CODE[1:10];
      INTEGER SUBNUM,L,TL,I,R,T,K,WHERE,N,TR,LR;
      BOOLEAN EOF;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  SUBNUM:=STRNGINT(GNXTLEX(EOF,3));
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    L:=DEFLFRAME(LOPNFRAME);
    ASSIGNTEMP(TL);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(TL);
    I:=MAKESPLADR(L,ORANK[L]+1,1);
    INSERTCODE(CODE,I,1);
    LOPNFRAME[1]:=LOPNFRAME[2]:=LOPNFRAME[3]:=I+1;
    LOPNFRAME[4]:=TL;
  END;
  R:=0;
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
         R:=DEFLFRAME(ROPNFRAME);
  IF LOPNFRAME[6]=INT(BINTYPE) AND (SCALAR(LOPNFRAME) OR
    (LOPNFRAME[7]=1 AND LOPNFRAME[8]=ROPNFRAME[7+SUBNUM]))
    AND SUBNUM =1 AND SUBNUM =ROPNFRAME[7]  THEN
  BEGIN
    I:=BNXTSPL(ROPNFRAME[2]);
    ASSIGNTEMP(T);
    CODE[1]:=CLRCODE OR BOOL(T);
    INSERTCODE(CODE,I,1);
    ASSIGNLREG(LR);  LNKREG[LR]:=LOPNFRAME[5];
    CODE[1]:=CRJCODE OR BOOL(LR);
    INSERTCODE(CODE,LOPNFRAME[2],1);
    K:=ROPNFRAME[7]-SUBNUM;
    WHERE:=ROPNFRAME[2];
    FOR I:=1 UNTIL K DO WHERE:=BNXTSPL(WHERE);
    INSERTCODE(CODE,WHERE,1);
    UPDATEPOINTER(ROPNFRAME,1,3,WHERE,1);
    CODE[1]:=TSTCODE OR BOOL(LOPNFRAME[4]);
    CODE[2]:=BNECODE OR %1;
```



```
    CODE[3]:=ENDMARK;
    CODE[4]:=INCCODE OR BOOL(T);
    IF R#0 THEN
    BEGIN
      N:=5;   ASSIGNTEMP(TR);   ROPNFRAME[4]:=TR;
      CODE[5]:=MOVCODE OR ATPISOURCE OR BOOL(TR);
    END
    ELSE N:=4;
    INSERTCODE(CODE,ROPNFRAME[2],N);
    UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],N);
    ROPNFRAME[7+SUBNUM]:=-(T+64*ROPNFRAME[7+SUBNUM]);
    PUSHFRAME(STACK,ROPNFRAME);
  END
  ELSE WRITE("[N]BAD ARG MIX TO COMPRESS");
END;



PROCEDURE EXPAND(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY MFRAME,LOPNFRAME,
              ROPNFRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER SUBNUM,L,I,TL,R,TR,T,LR1,LO,LR2;
      BOOLEAN EOF;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  SUBNUM:=STRNGINT(GNXTLEX(EOF,3));
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    L:=DEFLFRAME(LOPNFRAME);
    ASSIGNTEMP(TL);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(TL);
    I:=MAKESPLADR(L,ORANK[L]+1,1);
    INSERTCODE(CODE,I,1);
    LOPNFRAME[1]:=LOPNFRAME[2]:=LOPNFRAME[3]:=I+1;
    LOPNFRAME[4]:=TL;
  END;
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    R:=DEFLFRAME(ROPNFRAME);
    ASSIGNTEMP(TR);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(TR);
    INSERTCODE(CODE,ROPNFRAME[2],1);
    ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=
              ROPNFRAME[2]+1;
    ROPNFRAME[4]:=TR;
  END;
```

```
  IF LOPNFRAME[6]=INT(BINTYPE) AND (LOPNFRAME[7]=1 AND
      LOPNFRAME[8] =ROPNFRAME[SUBNUM+7]) AND SUBNUM =1
      AND SUBNUM  = ROPNFRAME[7]     THEN
  BEGIN
    I:=BNXTSPL(ROPNFRAME[2]);
    ASSIGNTEMP(T);
    CODE[1]:=CLRCODE OR BOOL(T);
    INSERTCODE(CODE,I,1);
    ASSIGNLREG(LR1);     LNKREG[LR1]:=LOPNFRAME[5];
    CODE[1]:=CRJCODE OR BOOL(LR1);
    INSERTCODE(CODE,LOPNFRAME[2],1);
    ASSIGNLNUM(LO,0);
    ORANK[LO]:=ROPNFRAME[7];
    OBETA[LO]:=0;
    OOPNTYPE[LO]:=ROPNFRAME[6];
    FOR I:=1 UNTIL ORANK[LO] DO
    BEGIN
      ORHO[LO,I]:=ROPNFRAME[I+7];  ODELTA[LO,I]:=0;
    END;
    ORHO[LO,SUBNUM]:=LOPNFRAME[8];
    GENSCODE(LO,ORANK[LO]);
    MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=
              MAKESPLADR(LO,ORANK[LO]+1,1);
    MFRAME[4]:=ROPNFRAME[4];  MFRAME[5]:=LO;
    MFRAME[6]:=ROPNFRAME[6];
    I:=MAKESPLADR( O,SUBNUM+1,1);
    INSERTCODE(CODE,I,1);
    UPDATEPOINTER(MFRAME,1,3,I,1);
    MFRAME[7]:=ORANK[LO];
    FOR I:=1 UNTIL MFRAME[7] DO MFRAME[I+7]:=ORHO[LO,I];
    ASSIGNLREG(LR2);     LNKREG[LR2]:=ROPNFRAME[5];
    CODE[1]:=CRJCODE OR BOOL(LR2);
    INSERTCODE(CODE,ROPNFRAME[2],1);
    CODE[1]:=CLRCODE OR BOOL(ROPNFRAME[4]);
    CODE[2]:=TSTCODE OR BOOL(LOPNFRAME[4]);
    CODE[3]:=BEQCODE OR %2;
    CODE[4]:=INCCODE OR BOOL(T);
    CODE[5]:=CRJCODE OR BOOL(LR2);
    INSERTCODE(CODE,MFRAME[2],5);
    UPDATEPOINTER(MFRAME,1,3,MFRAME[2],5);
    PUSHFRAME(STACK,MFRAME);
  END
  ELSE WRITE("[N]BAD ARG MIX TO EXPAND");
END;



PROCEDURE  CATENATE(STACK);
      INTEGER ARRAY STACK;
BEGIN
```



```
    INTEGER ARRAY MFRAME,LOPNFRAME,
          ROPNFRAME[1:STACK[0]];
    BOOLEAN ARRAY  CODE[1:10];
    INTEGER L,R,PRODR,PRODL,I,LO,T,LR1,LR2;

POPFRAME(STACK,LOPNFRAME);
POPFRAME(STACK,ROPNFRAME);
IF BOOL(LOPNFRAME[1]) AND LMASK THEN
          L:=DEFLFRAME(LOPNFRAME);
IF BOOL(ROPNFRAME[1]) AND LMASK THEN
          R:=DEFLFRAME(ROPNFRAME);
IF ROPNFRAME[7] =0 THEN
BEGIN
  PRODR:=1;
  FOR I:=1 UNTIL ROPNFRAME[7] DO
    PRODR:=IF ROPNFRAME[I+7] =0 THEN
          PRODR*ROPNFRAME[I+7] ELSE -1;
END
ELSE PRODR:=-1;
IF LOPNFRAME[7] =0 THEN
BEGIN
  PRODL:=1;
  FOR I:=1 UNTIL LOPNFRAME[7] DO
    PRODL:=IF LOPNFRAME[I+7] =0 THEN
          PRODL*LOPNFRAME[I+7] ELSE-1;
END
ELSE PRODL:=-1;
IF (PRODR=1 OR ROPNFRAME[7]=1) AND
    (PRODL=1 OR LOPNFRAME[7]=1)
    AND (LOPNFRAME[6]=ROPNFRAME[6]) THEN
BEGIN
  ASSIGNLNUM(LO,0);  ORANK[LO]:=1;  OBETA[LO]:=0;
  ODELTA[LO,1]:=1;   GENSCODE(LO,1);
  IF LOPNFRAME[8]#-1 AND ROPNFRAME[8]#-1 THEN
    ORHO[LO,1]:=LOPNFRAME[8]+ROPNFRAME[8]
  ELSE ORHO[LO,1]:=-1;
  OOPNTYPE[LO]:=LOPNFRAME[6];
  ASSIGNLREG(LR1);  LNKREG[LR1]:=LO;
  ASSIGNLREG(LR2);  LNKREG[LR2]:=ROPNFRAME[5];
  ASSIGNTEMP(T);
  CODE[1]:=MOVCODE OR BOOL(FLDSIZE*ROPNFRAME[4]) OR
          BOOL(T);
  CODE[2]:=CRJCODE OR BOOL(LR1);
  INSERTCODE(CODE,ROPNFRAME[2],2);
  CODE[1]:=MOVCODE OR BOOL(FLDSIZE*LOPNFRAME[4]) OR
          BOOL(T);
  INSERTCODE(CODE,LOPNFRAME[2],2);
  CODE[1]:=CRJCODE OR BOOL(LR1);
  I:=MAKESPLADR(LO,2,1);
  INSERTCODE(CODE,I,1);
  MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=I;
  I:=FNXTSPL(ROPNFRAME[2]);
  CODE[1]:=CRJCODE OR BOOL(LR2);
```

```
    INSERTCODE(CODE,I,1);
    I:=FNXTSPL(LOPNFRAME[2]);
    INSERTCODE(CODE,I,1);
    MFRAME[4]:=T;
    MFRAME[5]:=LOPNFRAME[5];
    MFRAME[6]:=LOPNFRAME[6];
    MFRAME[7]:=1;
    MFRAME[8]:=ORHO[LO,1];
    PUSHFRAME(STACK,MFRAME);
  END
  ELSE WRITE("[N]BAD ARG MIX TO CATENATE");
END;




PROCEDURE RAVEL(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY FRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER L,T,LR,LO,I,PROD;

  POPFRAME(STACK,FRAME);
  IF BOOL(FRAME[1]) AND LMASK THEN
  BEGIN
    L:=DEFLFRAME(FRAME);
    ASSIGNTEMP(T);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
    INSERTCODE(CODE,FRAME[2],1);
    FRAME[4]:=T;
    UPDATEPOINTER(FRAME,1,3,FRAME[2],1);
  END;
  ASSIGNLNUM(LO,0);
  ORANK[LO]:=1;
  IF FRAME[7] =0 THEN
  BEGIN
    PROD:=1;
    FOR I:=1 UNTIL FRAME[7] DO
      PROD:=IF FRAME[I+7] =0 THEN PROD*FRAME[I+7] ELSE -1;
  END
  ELSE PROD:=-1;
  ORHO[LO,1]:=PROD;  ODELTA[LO,1]:=0;  OBETA[LO]:=0;
  OOPNTYPE[LO]:=FRAME[6];
  GENSCODE(LO,1);
  ASSIGNLREG(LR);  LNKREG[LR]:=LO;
  CODE[1]:=CRJCODE OR BOOL(LR);
  INSERTCODE(CODE,FRAME[2],1);
  I:=MAKESPLADR(LO,2,1);
  INSERTCODE(CODE,I,1);
```



```
  FRAME[1]:=FRAME[2]:=FRAME[3]:=I;
  FRAME[7]:=1;
  FRAME[8]:=PROD;
  PUSHFRAME(STACK,FRAME);
END;




PROCEDURE DROP(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY MFRAME,LOPNFRAME,
              ROPNFRAME[1:STACK[0]],
              DROPVEC[1:10];
      BOOLEAN ARRAY CODE[1:10];
      BOOLEAN NULL;
      INTEGER LK,RK,L,I,T;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(LOPNFRAME[1]) AND BOOL(ROPNFRAME[1])
            AND LMASK THEN
  BEGIN
    LK:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    RK:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
    IF RANK[LK]=1 AND RHO[LK,1]=RANK[RK] AND
       INT(OPNTYPE[LK])=INT(INTTYPE) THEN
    BEGIN
      NULL:=FALSE;
      FOR I:=1 UNTIL RANK[RK] DO NULL:=NULL OR
            ABS(DATA[BETA[LK]+I-1])  = RHO[RK,I];
      IF NULL THEN WRITE("[N]DROP RESULT NULL ARRAY")
      ELSE
      BEGIN
        FOR I:=1 UNTIL RANK[RK] DO DROPVEC[I]:=
            DATA[BETA[LK]+I-1];
        ASSIGNLNUM(L,RK);
        LDROP(L,DROPVEC);
        GENSCODE(L,RANK[RK]);
        ASSIGNTEMP(T);
        CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
        I:=MAKESPLADR(L,ORANK[L]+1,1);
        INSERTCODE(CODE,I,1);
        MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=I+1;
        MFRAME[4]:=T;
        MFRAME[5]:=L;
        MFRAME[6]:=OOPNTYPE[L];
        MFRAME[7]:=ORANK[L];
        FOR I:=1 UNTIL MFRAME[7] DO
```

```
            MFRAME[I+7]:=ORHO[L,I];
          PUSHFRAME(STACK,MFRAME);
        END;
      END
      ELSE WRITE("[N]BAD LEFT ARG FOR DROP");
    END
    ELSE WRITE("[N]BOTH DROP ARGS MUST BE SIMPLE LADDERS");
END;




PROCEDURE TAKE(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY  MFRAME,LOPNFRAME,ROPNFRAME[1:STACK[0]],
              TAKEVEC[1:10];
      BOOLEAN ARRAY CODE[1:10];
      BOOLEAN NULL,EXTEND;
      INTEGER LK,RK,I,PRODR,PRODL,L,T;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(LOPNFRAME[1]) AND BOOL(ROPNFRAME[1])
      AND LMASK THEN
  BEGIN
    LK:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    RK:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
    PRODL:=1;
    FOR I:=1 UNTIL RANK[LK] DO PRODL:=PRODL*RHO[LK,I];
    PRODR:=1;
    FOR I:=1 UNTIL RANK[RK] DO PRODR:=PRODR*RHO[RK,I];
    IF PRODL=1 AND (PRODR=1 OR RANK[RK]=1) OR
       RANK[LK]=1 AND (PRODR=1 OR RHO[LK,1]=RANK[RK])
       AND INT(OPNTYPE[LK])=INT(INTTYPE) THEN
    BEGIN
      NULL:=FALSE;
      FOR I:=1 UNTIL RHO[LK,1] DO NULL:=NULL OR
              DATA[BETA[LK]+I-1]=0;
      IF NULL THEN WRITE("[N]TAKE RESULT NULL ARRAY")
      ELSE
      BEGIN
        EXTEND:=FALSE;
        FOR I:=1 UNTIL RHO[LK,1] DO EXTEND:=EXTEND OR
                ABS(DATA[BETA[LK]+I-1])   RHO[RK,I];
        IF EXTEND THEN WRITE("[N]NO EXTEND")
        ELSE
        BEGIN
          FOR I:=1 UNTIL RHO[LK,1] DO
              TAKEVEC[I]:=DATA[BETA[LK]+I-1];
          ASSIGNLNUM(L,RK);
```



```
          LTAKE(L,TAKEVEC);
          GENSCODE(L,RANK[RK]);
          ASSIGNTEMP(T);
          CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
          I:=MAKESPLADR(L,ORANK[L]+1,1);
          INSERTCODE(CODE,I,1);
          MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=I+1;
          MFRAME[4]:=T;        MFRAME[5]:=L;
          MFRAME[6]:=OOPNTYPE[L];
          MFRAME[7]:=ORANK[L];
          FOR I:=1 UNTIL MFRAME[7] DO
            MFRAME[I+7]:=ORHO[L,I];
          PUSHFRAME(STACK,MFRAME);
        END;
      END;
    END
    ELSE WRITE("[N]BAD ARG MIX FOR TAKE");
  END
  ELSE WRITE("[N]BOTH TAKE ARGS MUST BE SIMPLE LADDERS");
END;




PROCEDURE MONADTRANSPOSE(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY FRAME[1:STACK[0]],TVEC[1:10];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER I,L,T;

  POPFRAME(STACK,FRAME);
  IF BOOL(FRAME[1]) AND LMASK THEN
  BEGIN
    L:=DEFLFRAME(FRAME);
    IF ORANK[L]  = 2 THEN
    BEGIN
      FOR I:=1 UNTIL ORANK[L]-2 DO TVEC[I]:=I;
      TVEC[ORANK[L]-1]:=ORANK[L];
      TVEC[ORANK[L]]:=ORANK[L]-1;
      LTRANSPOSE(L,TVEC);
    END;
    ASSIGNTEMP(T);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
    I:=MAKESPLADR(L,ORANK[L]+1,1);
    INSERTCODE(CODE,I,1);
    FRAME[1]:=FRAME[2]:=FRAME[3]:=I+1;
    FRAME[4]:=T;
    FOR I:=1 UNTIL ORANK[L] DO FRAME[I+7]:=ORHO[L,I];
    PUSHFRAME(STACK,FRAME);
  END
  ELSE WRITE("[N]ARG TO M. TRANSPOSE NOT SIMPLE LADDER");
```

```
END;


PROCEDURE DYADTRANSPOSE(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY LOPNFRAME,
          ROPNFRAME[1:STACK[0]],TVEC[1:10];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER RK,LK,R,I,T;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(LOPNFRAME[1]) AND BOOL(ROPNFRAME[1])
     AND LMASK THEN
  BEGIN
    RK:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
    LK:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    IF RANK[LK]=1 AND RHO[LK,1]=RANK[RK] AND
       INT(OPNTYPE[LK])=INT(INTTYPE) THEN
    BEGIN
      ASSIGNLNUM(R,RK);
      FOR I:=1 UNTIL RANK[RK] DO
              TVEC[I]:=DATA[BETA[LK]+I-1];
      LTRANSPOSE(R,TVEC);
      GENSCODE(R,ORANK[R]);
      ASSIGNTEMP(T);
      CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
      I:=MAKESPLADR(R,ORANK[R]+1,1);
      INSERTCODE(CODE,I,1);
      ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=I+1;
      ROPNFRAME[4]:=T;
      ROPNFRAME[5]:=R;
      ROPNFRAME[6]:=OOPNTYPE[R];
      ROPNFRAME[7]:=ORANK[R];
      FOR I:=1 UNTIL ROPNFRAME[7] DO
            ROPNFRAME[I+7]:=ORHO[R,I];
      PUSHFRAME(STACK,ROPNFRAME);
    END
    ELSE WRITE("[N]BAD ARG MIX FOR DYADIC TRANSPOSE");
  END
  ELSE WRITE("[N]ARG OF D TRANSPOSE NOT SIMPLE LDRS");
END;




PROCEDURE SHAPE(STACK);
      INTEGER ARRAY STACK;
```



```
BEGIN
      INTEGER ARRAY FRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER K,I,T,L;
      STRING A,B;

  POPFRAME(STACK,FRAME);
  IF BOOL(FRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(FRAME[1]) AND SYMBMASK);
    IF NOPNS   MAXOPNS AND
            NWORDS+RANK[K]  = MAXWORDS THEN
    BEGIN
      NOPNS:=NOPNS+1;
      A:="%";          B:=INTSTRNG(NOPNS);
      OPNNAME[NOPNS]:=MCONCAT(A,B);
      OPNTYPE[NOPNS]:=INTTYPE;
      BETA[NOPNS]:=NWORDS+1;
      RANK[NOPNS]:=1;
      RHO[NOPNS,1]:=RANK[K];
      FOR I:=1 UNTIL RANK[K] DO DATA[NWORDS+I]:=RHO[K,I];
      NWORDS:=NWORDS+RANK[K];
      ASSIGNLNUM(L,0);
      ORANK[L]:=1;     ORHO[L,1]:=RANK[K];
      ODELTA[L,1]:=1;  OOPNTYPE[L]:=INT(INTTYPE);
      OBETA[L]:=BETA[NOPNS];
      GENSCODE(L,1);
      ASSIGNTEMP(T);
      CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
      I:=MAKESPLADR(L,2,1);
      INSERTCODE(CODE,I,1);
      FRAME[1]:=FRAME[2]:=FRAME[3]:=I+1;
      FRAME[4]:=T;
      FRAME[5]:=L;     FRAME[6]:=INT(INTTYPE);
      FRAME[7]:=1;     FRAME[8]:=RANK[K];
      PUSHFRAME(STACK,FRAME);
    END
    ELSE WRITE("[N]CAN'T CREATE NEW VARIABLE FOR SHAPE");
  END
  ELSE WRITE("[N]ARGUMENT OF SHAPE MUST BE SIMPLE LADDER");
END;



PROCEDURE RESTRUCTURE(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY MFRAME,LOPNFRAME,ROPNFRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER L,I,K,T,LR,R;
```

```
  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    IF RANK[K]=1 AND RHO[K,1]  = MAXRANK THEN
    BEGIN
      IF BOOL(ROPNFRAME[1]) AND LMASK THEN
      BEGIN
        R:=DEFLFRAME(ROPNFRAME);
        ASSIGNTEMP(T);
        CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
        INSERTCODE(CODE,ROPNFRAME[2],1);
        UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],1);
        ROPNFRAME[4]:=T;
      END;
      ASSIGNLNUM(L,0);
      ORANK[L]:=RHO[K,1];
      FOR I:=1 UNTIL ORANK[L] DO
      BEGIN
        ORHO[L,I]:=DATA[BETA[K]+I-1];  ODELTA[L,I]:=1;
      END;
      OOPNTYPE[L]:=INT(INTTYPE);
      GENSCODE(L,ORANK[L]);
      ASSIGNLREG(LR);  LNKREG[LR]:=ROPNFRAME[5];
      CODE[1]:=CRJCODE OR BOOL(LR);
      I:=MAKESPLADR(L,ORANK[L]+1,1);
      INSERTCODE(CODE,I,1);
      INSERTCODE(CODE,ROPNFRAME[2],1);
      MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=I+1;
      MFRAME[4]:=ROPNFRAME[4];
      MFRAME[5]:=L;
      MFRAME[6]:=ROPNFRAME[6];
      MFRAME[7]:=ORANK[L];
      FOR I:=1 UNTIL ORANK[L] DO MFRAME[I+7]:=ORHO[L,I];
      PUSHFRAME(STACK,MFRAME);
    END
    ELSE WRITE("[N]BAD LEFT ARG TO RESTRUCTURE");
  END
  ELSE WRITE("[N]L ARG TO RESTRUCTURE MUST BE SIMPLE LADDE
END;



PROCEDURE INDEXGEN(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY ROPNFRAME[1:STACK[0]];
```



```
      INTEGER K,I,PROD,L,LR,T;
      BOOLEAN ARRAY CODE[1:10];
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
    PROD:=1;
    FOR I:=1 UNTIL RANK[K] DO PROD:=PROD*RHO[K,I];
    IF PROD=1 AND INT(OPNTYPE[K])=INT(INTTYPE) THEN
    BEGIN
      ASSIGNLNUM(L,0);
      ORANK[L]:=1;  ORHO[L,1]:=DATA[BETA[K]];
      ODELTA[L,1]:=1;
      OOPNTYPE[L]:=INT(INTTYPE);
      GENSCODE(L,1);    ASSIGNTEMP(T);
      CODE[1]:=MOVCODE OR BOOL(FLDSIZE*8) OR BOOL(T);
      I:=MAKESPLADR(L,2,1);
      INSERTCODE(CODE,I,1);
      ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=I+1;
      ROPNFRAME[4]:=T;              ROPNFRAME[5]:=L;
      ROPNFRAME[6]:=INT(INTTYPE);  ROPNFRAME[7]:=1;
      ROPNFRAME[8]:=ORHO[L,1];
      PUSHFRAME(STACK,ROPNFRAME);
    END
    ELSE WRITE("[N]ILLEGAL ARG TO .IOM (BAD SIMPLE LADDER)");
  END
  ELSE WRITE("[N]ARG OF .IOM MUST BE A SIMPLE LADDER");
END;



PROCEDURE INDEXOF(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY LOPNFRAME,ROPNFRAME,
        MFRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];
      INTEGER OUT,T1,I,LR,RT,RI,LI;
      BOOLEAN SOURCE;

  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    RI:=DEFLFRAME(ROPNFRAME);
    ASSIGNTEMP(RT);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(RT);
    I:=MAKESPLADR(RI,ORANK[RI]+1,1);
    INSERTCODE(CODE,I,1);
    ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=I+1;
    ROPNFRAME[4]:=RT;
  END;
```

```
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
              LI:=DEFLFRAME(LOPNFRAME);
  IF LOPNFRAME[7]=1 THEN
  BEGIN
    ASSIGNLREG(LR);  LNKREG[LR]:=LOPNFRAME[5];
    ASSIGNTEMP(T1);
    CODE[1]:=CRJCODE OR BOOL(LR);
    INSERTCODE(CODE,ROPNFRAME[1],1);
    UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[1],1);
    OUT:=I:=FNXTSPL(LOPNFRAME[2]);
    INSERTCODE(CODE,I,1);
    I:=BNXTSPL(LOPNFRAME[2]);
    SOURCE:=BOOL(FLDSIZE*(6+4*SPLADR(I)));
    CODE[1]:=MOVCODE OR SOURCE OR BOOL(T1);
    CODE[2]:=INCCODE OR BOOL(T1);
    INSERTCODE(CODE,I,2);
    CODE[1]:=CMPCODE OR BOOL(FLDSIZE*LOPNFRAME[4]) OR
                         BOOL(ROPNFRAME[4]);
    CODE[2]:=BNECODE OR %2;
    SOURCE:=BOOL(FLDSIZE*(4+4*SPLADR(I)));
    CODE[3]:=MOVCODE OR SOURCE OR BOOL(T1);
    SOURCE:=BOOL(FLDSIZE*(6+4*SPLADR(I)));
    CODE[4]:=MOVCODE OR SOURCE OR BOOL(4+4*SPLADR(I));
    INSERTCODE(CODE,LOPNFRAME[2],4);
    ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=OUT;
    ROPNFRAME[4]:=T1;
    ROPNFRAME[6]:=INT(INTTYPE);
    PUSHFRAME(STACK,ROPNFRAME);
  END
  ELSE WRITE("[N]LEFT OPERAND OF .IO NOT A VECTOR");
END;



PROCEDURE MEMBERSHIP(STACK);
      INTEGER ARRAY STACK;
BEGIN
      INTEGER ARRAY LOPNFRAME,ROPNFRAME,
              MFRAME[1:STACK[0]];
      INTEGER RI,LI,LT,I,K,T1,LR;
      BOOLEAN ARRAY  CODE[1:15];
  POPFRAME(STACK,LOPNFRAME);
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
              RI:=DEFLFRAME(ROPNFRAME);
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    LI:=DEFLFRAME(LOPNFRAME);
    ASSIGNTEMP(LT);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(LT);
    I:=MAKESPLADR(LI,ORANK[LI]+1,1);
    INSERTCODE(CODE,I,1);
```



```
    LOPNFRAME[1]:=LOPNFRAME[2]:=LOPNFRAME[3]:=I+1;
    LOPNFRAME[4]:=LT;
  END;
  ASSIGNLREG(LR);   ASSIGNTEMP(T1);
  LNKREG[LR]:=ROPNFRAME[5];
  CODE[1]:=CRJCODE OR BOOL(LR);
  I:=ROPNFRAME[5]; K:=2*ORANK[I]+1;
  I:=MAKESPLADR(I,K,IMAX(INT(SCODE[I,K,0]),1));
  INSERTCODE(CODE,I,1);
  CODE[1]:=CLRCODE OR BOOL(T1);
  CODE[2]:=CRJCODE OR BOOL(LR);
  INSERTCODE(CODE,LOPNFRAME[2],2);
  UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],2);
  CODE[1]:=CMPCODE OR BOOL(FLDSIZE*ROPNFRAME[4]) OR
                      BOOL(LOPNFRAME[4]);
  CODE[2]:=BNECODE OR BOOL(ROPNFRAME[7]+1);
  CODE[3]:=INCCODE OR BOOL(T1);
  FOR I:= 1 UNTIL ROPNFRAME[7] DO
      CODE[I+3]:= MOVCODE OR BOOL(FLDSIZE*(6+I*4))
                      OR BOOL(4+I*4);
  INSERTCODE(CODE,ROPNFRAME[2],3+ROPNFRAME[7]);
  MFRAME[1]:=MFRAME[2]:=MFRAME[3]:=LOPNFRAME[2];
  MFRAME[4]:=T1;
  MFRAME[5]:=LOPNFRAME[5];
  MFRAME[6]:=INT(BINTYPE);
  FOR I:= 0 UNTIL LOPNFRAME[7] DO
    MFRAME[I+7]:=LOPNFRAME[I+7];
  PUSHFRAME(STACK,MFRAME);
END;



PROCEDURE SCANINIT(L,S,OP,T2);
      INTEGER L,S,OP,T2;          VALUE OP,T2;

BEGIN  INTEGER WHERE,T3;   BOOLEAN ARRAY CODE[1:1];
  IF (WHERE:=BNXTSPL(MAKESPLADR(L,S,1)))=0  THEN
      WRITE("[N]SCAN OF A SCALAR")
  ELSE
  BEGIN
    IF OPRTYPE[OP] AND NOCOMMUTEMASK OR
            OPRIDENT[OP]=0 THEN
          CODE[1]:=CLRCODE OR BOOL(T2)
    ELSE
    BEGIN
      ASSIGNTEMP(T3);
      TINITLIST[ILP]:=T3;
      TINITLIST[ILP+1]:=OPRIDENT[OP];
      ILP:=ILP+2;
      CODE[1]:=MOVCODE OR BOOL(T3*FLDSIZE) OR BOOL(T2);
    END;
```

```
    INSERTCODE(CODE,WHERE,1);
    L:=LADR(WHERE);     S:=SPLADR(WHERE);
  END;
END;



PROCEDURE REDUCINIT(L,S,OP,T1,T2);
      INTEGER L,S,OP,T1,T2;      VALUE OP,T1;

BEGIN  INTEGER WHERE,J,RNK,T3;   BOOLEAN ARRAY CODE[1:1];
  IF (WHERE:=BNXTSPL(MAKESPLADR(L,S,1)))=0 THEN
       WRITE("[N]REDUCTION OF A SCALAR")
  ELSE
  BEGIN
    IF OPRIDENT[OP] = 0 THEN CODE[1]:= CLRCODE OR BOOL(T1)
    ELSE
    BEGIN
      ASSIGNTEMP(T3);
      TINITLIST[ILP]:=T3;  TINITLIST[ILP+1]:=OPRIDENT[OP];
      ILP:=ILP+2;
      CODE[1]:=MOVCODE OR BOOL(T3*FLDSIZE) OR BOOL(T1);
    END;
    INSERTCODE(CODE,WHERE,1);
    IF NOCOMMUTEMASK AND OPRTYPE[OP] THEN
    BEGIN
      ASSIGNTEMP(T2);
      CODE[1]:=CLRCODE OR BOOL(T2);
      INSERTCODE(CODE,WHERE,1);
    END;
    L:=LADR(WHERE);   S:=SPLADR(WHERE);
  END;
END;



PROCEDURE SCANACCUM(ROPNFRAME,OP,T1,T2);
      INTEGER ARRAY ROPNFRAME;  INTEGER OP,T1,T2;
                                VALUE OP,T1,T2;
BEGIN
      INTEGER ARRAY LA[1:4];  BOOLEAN ARRAY CODE[1:10];
      INTEGER N,T3,T4;

  IF NOCOMMUTEMASK AND OPRTYPE[OP] THEN
  BEGIN
    MAKESCODE(CODE,ROPNFRAME,OP,T1,T2);
    N:=NSINSTRS[OP];
    ROPNFRAME[4]:=T1;
  END
  ELSE
  BEGIN
```



```
    LA[4]:=ROPNFRAME[4];
    ROPNFRAME[4]:=T1;
    MAKECODE(CODE,LA,ROPNFRAME,OP,T3);
    N:=NINSTRS[OP];
  END;
  ASSIGNTEMP(T4);
  CODE[N+1]:=MOVCODE OR BOOL(FLDSIZE*ROPNFRAME[4])
                     OR BOOL(T4);
  INSERTCODE(CODE,ROPNFRAME[2],N+1);
  UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],N+1);
  ROPNFRAME[4]:=T4;
END;



PROCEDURE REDUCACCUM(ROPNFRAME,OP,T1,T2);
      INTEGER ARRAY ROPNFRAME;  INTEGER OP,T1,T2;
      VALUE OP,T1,T2;

BEGIN  INTEGER ARRAY LA[1:4];    BOOLEAN ARRAY CODE[1:10];
       INTEGER N;
  LA[4]:=ROPNFRAME[4];
  ROPNFRAME[4]:=T1;
  N:=NINSTRS[OP];
  IF NOCOMMUTEMASK AND OPRTYPE[OP] THEN
  BEGIN
    MAKERCODE(CODE,LA,ROPNFRAME,OP,T1,T2);
    N:=NRINSTRS[OP];
  END
  ELSE MAKECODE(CODE,LA,ROPNFRAME,OP,T1);
  INSERTCODE(CODE,ROPNFRAME[2],N);
  UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],N);
END;



PROCEDURE REDUC(STACK);  INTEGER ARRAY STACK;

BEGIN  INTEGER OP,L,S,RI,RT,RNK,I,K,T1,T2;
       INTEGER ARRAY ROPNFRAME[1:STACK[0]];
       BOOLEAN SYMB;
       BOOLEAN ARRAY CODE[1:10];
  IF NEXTSYMB(SYMB,3) THEN
  BEGIN
    IF SYMB AND OPNMASK THEN
      WRITE("[N]NO OPERATOR WITH REDUCTION")
    ELSE
```

```
    BEGIN
      OP:=INT(SYMB AND SYMBMASK);
      IF OPRTYPE[OP] AND DYADSCALMASK THEN
      BEGIN
        POPFRAME(STACK,ROPNFRAME);
        IF BOOL(ROPNFRAME[1]) AND LMASK THEN
             RI:=DEFLFRAME(ROPNFRAME);
        L:=LADR(ROPNFRAME[2]);
        S:=SPLADR(ROPNFRAME[2]);
        ASSIGNTEMP(T1);
        REDUCINIT(L,S,OP,T1,T2);  !CAN CHANGE L AND S;
                                  !ASSIGNS A VALUE TO T2;
                                  !IF IT IS REQUIRED;
        REDUCACCUM(ROPNFRAME,OP,T1,T2);
        S:=2*ORANK[L]+2-S;
        ROPNFRAME[1]:=MAKESPLADR(L,S,1);
        ROPNFRAME[2]:=ROPNFRAME[1];
        ROPNFRAME[7]:=ROPNFRAME[7]-1;
        PUSHFRAME(STACK,ROPNFRAME);
      END
      ELSE WRITE
           ("[N]REDUCTION WITH NON DYADICSCAL OPERATOR");
    END;
  END
  ELSE WRITE("[N]NO SYMBOL WITH REDUCTION");
END;


PROCEDURE SCAN(STACK);  INTEGER ARRAY STACK;

BEGIN  INTEGER OP,L,S,RI,RT,RNK,I,K,T1,T2,N;
       INTEGER ARRAY ROPNFRAME[1:STACK[0]];
       BOOLEAN SYMB;
       BOOLEAN ARRAY CODE[1:10];

  IF NEXTSYMB(SYMB,3) THEN
  BEGIN
    IF SYMB AND OPNMASK THEN
      WRITE("[N]NO OPERATOR WITH SCAN")
    ELSE
    BEGIN
      OP:=INT(SYMB AND SYMBMASK);
      IF OPRTYPE[OP] AND DYADSCALMASK THEN
      BEGIN
        POPFRAME(STACK,ROPNFRAME);
        IF BOOL(ROPNFRAME[1]) AND LMASK THEN
             RI:=DEFLFRAME(ROPNFRAME);
        L:=LADR(ROPNFRAME[2]);
        S:=SPLADR(ROPNFRAME[2]);
        ASSIGNTEMP(T1);
        IF NOCOMMUTEMASK AND OPRTYPE[OP] THEN
        BEGIN
          ASSIGNTEMP(T2);
```



```
          SCANINIT(L,S,OP,T2);          !CHANGES L AND S;
        END
        ELSE SCANINIT(L,S,OP,T1);       !CHANGES L AND S;
        SCANACCUM(ROPNFRAME,OP,T1,T2);
        PUSHFRAME(STACK,ROPNFRAME);
      END
      ELSE WRITE("[N]SCAN WITH NON DYADICSCALAR OPERATOR");
    END;
  END
  ELSE WRITE("[N]NO SYMBOL WITH SCAN");
END;


PROCEDURE INNERPROD(STACK);  INTEGER ARRAY STACK;
BEGIN
      BOOLEAN SYMB;
      INTEGER LOPR,ROPR,I,J,K,LI1,LI2,RI,
              RNK,LRNK1,LRNK2,T1,T2,T3;
      INTEGER L,S,LR1,LR2,TEMPOUT,T4;
      INTEGER ARRAY TVEC[1:10],LOPNFRAME,
              ROPNFRAME[1:STACK[0]];
      BOOLEAN ARRAY CODE[1:10];

  IF NEXTSYMB(SYMB,3) THEN
  BEGIN
    IF SYMB AND OPNMASK THEN
      WRITE("[N]FIRST LEXIEM AFTER IP NOT OPR")
    ELSE
    BEGIN
      ROPR:=INT(SYMB AND SYMBMASK);
      IF NEXTSYMB(SYMB,3) THEN
      BEGIN
        IF SYMB AND OPNMASK THEN
          WRITE("[N]SECOND LEXIEM AFTER IP NOT OPR")
        ELSE
        BEGIN
          LOPR:=INT(SYMB AND SYMBMASK);
          POPFRAME(STACK,LOPNFRAME);
          POPFRAME(STACK,ROPNFRAME);
          IF BOOL(ROPNFRAME[1]) AND BOOL(LOPNFRAME[1])
              AND LMASK THEN
          BEGIN
            K:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
            J:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
            IF IPCONFORMCHK(K,J,LOPR,ROPR) THEN
            BEGIN
              ASSIGNLNUM(LI1,K);  ASSIGNLNUM(RI,J);
              RNK:=RANK[J];
              GENSCODE(RI,RNK);
              IF ORANK[RI] 1 THEN
              BEGIN
                TVEC[1]:=RANK[J];
```

```
  FOR I:=1 UNTIL RANK[J]-1 DO TVEC[I+1]:=I;
  LTRANSPOSE(RI,TVEC);
END;
IF ORANK[LI1] 1 THEN
BEGIN
  IF 0=(LI2:=LSPLIT(LI1,ORANK[LI1]-1)) THEN
          WRITE("[N]CAN'T SPLIT LADDER");
  ASSIGNTEMP(T1);  ASSIGNLREG(LR1);
  LRNK1:=RANK[K]-1;
  OUTPRODLNK[RI]:=LI1;
  GENSCODE(LI1,LRNK1);     GENSCODE(LI2,1);
  LNKREG[LR1]:=RI;
  CODE[1]:=MOVCODE OR PISOURCE OR BOOL(T1);
  CODE[2]:=CRJCODE OR BOOL(LR1);
  I:=MAKESPLADR(LI1,ORANK[LI1]+1,1);
  INSERTCODE(CODE,I,2);
  CODE[1]:=CRJCODE OR BOOL(LR1);
  I:=MAKESPLADR(RI,2*ORANK[RI]+1,1);
  INSERTCODE(CODE,I,1);
  CODE[1]:=MOVCODE OR BOOL(FLDSIZE*T1)
        OR BETADEST;
  I:=MAKESPLADR(LI2,1,1);
  INSERTCODE(CODE,I,1);
  ROPNFRAME[5]:=LI1;
END
ELSE
BEGIN
  ROPNFRAME[5]:=RI;  LI2:=LI1;
  GENSCODE(LI2,ORANK[LI2]);
END;

ASSIGNTEMP(T2); ASSIGNTEMP(T3);
ASSIGNLREG(LR2);
LNKREG[LR2]:=LI2;
L:=RI;  S:=ORANK[RI]+1;
REDUCINIT(L,S,LOPR,T3,T4);
CODE[1]:= MOVCODE OR ATPISOURCE OR BOOL(T2);
CODE[2]:= CRJCODE OR BOOL(LR2);
I:=MAKESPLADR(RI,ORANK[RI]+1,1);
INSERTCODE(CODE,I,2);
CODE[1]:= CRJCODE OR BOOL(LR2);
I:=MAKESPLADR(LI2,2,1);
INSERTCODE(CODE,I,1);
LOPNFRAME[4]:=0;  ROPNFRAME[4]:=TEMPOUT:=T2;
IF OPRTYPE[ROPR] AND ALLOCTEMPOUT THEN
        ASSIGNTEMP(TEMPOUT);
MAKECODE(CODE,LOPNFRAME,ROPNFRAME,ROPR,
                                  TEMPOUT);
ROPNFRAME[2]:=MAKESPLADR(LI2,2,1);
INSERTCODE(CODE,ROPNFRAME[2],NINSTRS[ROPR]);
UPDATEPOINTER(ROPNFRAME,2,2,ROPNFRAME[2],
                          NINSTRS[ROPR]);
REDUCACCUM(ROPNFRAME,LOPR,T3,T4);
```



```
          ROPNFRAME[1]:=MAKESPLADR(RI,ORANK[RI]+2,1);
        ROPNFRAME[3]:=ROPNFRAME[2]:=ROPNFRAME[1];
          ROPNFRAME[4]:=T3;
      ROPNFRAME[6]:=INT(OPNTYPE[J]);
          ROPNFRAME[7]:=IMAX(0,RANK[K]+RANK[J]-2);
          FOR I:=1 UNTIL RANK[K]-1 DO
                        ROPNFRAME[7+I]:=RHO[K,I];
          FOR I:=2 UNTIL RANK[J] DO
                    ROPNFRAME[5+RANK[K]+I]:=RHO[J,I];
          PUSHFRAME(STACK,ROPNFRAME);
        END
        ELSE WRITE("[N]BAD OPN OR OPR FOR IP");
      END
      ELSE WRITE("[N]BOTH IP OPNS NOT SIMPLE LADDERS");
    END;
  END
  ELSE WRITE("[N]NOT ENOUGH LEXIEMS AFTER IP");
 END;
 END
 ELSE WRITE("[N]NO LEXIEMS AFTER IP");
END;
```

```
PROCEDURE OUTPROD(STACK);      INTEGER ARRAY STACK;

BEGIN    INTEGER OP,LI,LT,RI,RT,LR,RRNK,LRNK,I,K,TEMPOUT;
         INTEGER ARRAY MFRAME,ROPNFRAME,
                       LOPNFRAME[1:STACK[0]];
         BOOLEAN SYMB;
         BOOLEAN ARRAY CODE[1:10];
  IF NEXTSYMB(SYMB,3) THEN
  BEGIN
    IF SYMB AND OPNMASK THEN
      WRITE("[N]NO OPERATOR FOLLOWING OUTERPRODUCT")
    ELSE
    BEGIN
      OP:=INT(SYMB AND SYMBMASK);
      IF OPRTYPE[OP] AND DYADSCALMASK THEN
      BEGIN
        POPFRAME(STACK,LOPNFRAME);
        POPFRAME(STACK,ROPNFRAME);
        IF BOOL(LOPNFRAME[1]) AND LMASK THEN
        BEGIN
          LI:=DEFLFRAME(LOPNFRAME);
          ASSIGNTEMP(LT);
          CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(LT);
```

```
          I:=MAKESPLADR(LI,ORANK[LI]+1,1);
          INSERTCODE(CODE,I,1);
          LOPNFRAME[1]:=LOPNFRAME[2]:=LOPNFRAME[3]:=I+1;
          LOPNFRAME[4]:=LT;
        END;
        IF BOOL(ROPNFRAME[1]) AND LMASK THEN
        BEGIN
          RI:=DEFLFRAME(ROPNFRAME);
          ASSIGNTEMP(RT);
          CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(RT);
          I:=MAKESPLADR(RI,ORANK[RI]+1,1);
          INSERTCODE(CODE,I,1);
          ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=I+1;
          ROPNFRAME[4]:=RT;
        END;
        OUTPRODLNK[LADR(ROPNFRAME[2])]:=LADR(LOPNFRAME[2]);
        ASSIGNLREG(LR);
        LNKREG[LR]:=ROPNFRAME[5];
        CODE[1]:=CRJCODE OR BOOL(LR);
        INSERTCODE(CODE,LOPNFRAME[1],1);
        UPDATEPOINTER(ROPNFRAME,1,3,LOPNFRAME[1],1);
        UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[1],1);
        TEMPOUT:=ROPNFRAME[4];
        IF OPRTYPE[OP] AND ALLOCTEMPOUT THEN
              ASSIGNTEMP(TEMPOUT);
        MAKECODE(CODE,LOPNFRAME,ROPNFRAME,OP,TEMPOUT);
        INSERTCODE(CODE,ROPNFRAME[2],NINSTRS[OP]);
        UPDATEPOINTER(LOPNFRAME,1,3,ROPNFRAME[2],
                              NINSTRS[OP]);
        UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],
                              NINSTRS[OP]);
        K:=ROPNFRAME[5];
        I:=MAKESPLADR(K,2*ORANK[K]+1,1);
        CODE[1]:=CRJCODE OR BOOL(LR);
        INSERTCODE(CODE,I,1);
        MERGEOPFRAME(MFRAME,LOPNFRAME,ROPNFRAME,TEMPOUT);
        PUSHFRAME(STACK,MFRAME);
      END
      ELSE WRITE("[N]OP NOT FOLLOWED BY BIN SCAL OP");
    END;
  END
  ELSE WRITE("[N]OP NOT FOLLOWED BY OPERATOR");
END;


PROCEDURE DYADSCAL(STACK,OP);  INTEGER OP; VALUE OP;
              INTEGER ARRAY STACK;

BEGIN    INTEGER RI,LI,RT,LT,LR,RRNK,LRNK,I,K,T,T1,TEMPOUT;
         INTEGER ARRAY ROPNFRAME,LOPNFRAME,
               MFRAME[1:STACK[0]];
         BOOLEAN ARRAY CODE[1:10];
```



```
         BOOLEAN SCALARR,SCALARL;

POPFRAME(STACK,LOPNFRAME);
POPFRAME(STACK,ROPNFRAME);
SCALARR:=SCALAR(ROPNFRAME);
SCALARL:=SCALAR(LOPNFRAME);
IF SCALARR AND SCALARL THEN
    WRITE("[N]TWO SCALARS TO DYADIC SCALAR OP")
ELSE IF SCALARL THEN
BEGIN
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    RI:=DEFLFRAME(ROPNFRAME);
    ASSIGNTEMP(T);     ROPNFRAME[4]:=T;
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
    INSERTCODE(CODE,ROPNFRAME[2],1);
    UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],1);
  END;
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    ASSIGNTEMP(T);
    IF DATA[BETA[K]] = 0 THEN
                  CODE[1]:=CLRCODE OR BOOL(T)
    ELSE
    BEGIN
      ASSIGNTEMP(T1);      TINITLIST[ILP]:=T1;
      TINITLIST[ILP+1]:=DATA[BETA[K]];     ILP:=ILP+2;
      CODE[1]:=MOVCODE OR BOOL(FLDSIZE*T1) OR BOOL(T);
    END;
    INSERTCODE(CODE,ROPNFRAME[2],1);
    UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],1);
    LOPNFRAME[4]:=T;
  END
  ELSE
  BEGIN
    ASSIGNLREG(LR);        LNKREG[LR]:=LOPNFRAME[5];
    CODE[1]:=CRJCODE OR BOOL(LR);
    INSERTCODE(CODE,LOPNFRAME[2],1);
    UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],1);
    I:=MAKESPLADR(LADR(ROPNFRAME[2]),1,1);
    INSERTCODE(CODE,I,1);
    UPDATEPOINTER(LOPNFRAME,1,3,I,1);
  END;
  IF OPRTYPE[OP] AND ALLOCTEMPOUT THEN
          ASSIGNTEMP(TEMPOUT)
       ELSE TEMPOUT:=ROPNFRAME[4];
  MAKECODE(CODE,LOPNFRAME,ROPNFRAME,OP,TEMPOUT);
  INSERTCODE(CODE,ROPNFRAME[2],NINSTRS[OP]);
  UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],NINSTRS[OP]);
  FOR I:=1 UNTIL 7+ROPNFRAME[7] DO
          MFRAME[I]:=ROPNFRAME[I];
  MFRAME[4]:=TEMPOUT;
```

```
  PUSHFRAME(STACK,MFRAME);
END
ELSE IF SCALARR THEN
BEGIN
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
                    LI:=DEFLFRAME(LOPNFRAME);
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(ROPNFRAME[1]) AND SYMBMASK);
    ASSIGNTEMP(T);          ROPNFRAME[4]:=T;
    IF DATA[BETA[K]] = 0 THEN
            CODE[1]:=CLRCODE OR BOOL(T)
    ELSE
    BEGIN
      ASSIGNTEMP(T1);
      TINITLIST[ILP]:=T1;
      TINITLIST[ILP+1]:=DATA[BETA[K]];      ILP:=ILP+2;
      CODE[1]:=MOVCODE OR BOOL(FLDSIZE*T1) OR BOOL(T);
    END;
    INSERTCODE(CODE,LOPNFRAME[2],1);
    UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],1);
  END
  ELSE
  BEGIN
    ASSIGNLREG(LR);         LNKREG[LR]:=ROPNFRAME[5];
    CODE[1]:=CRJCODE OR BOOL(LR);
    INSERTCODE(CODE,ROPNFRAME[2],1);
    UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],1);
    I:=MAKESPLADR(LADR(OPNFRAME[2]),1,1);
    INSERTCODE(CODE,I,1);
    UPDATEPOINTER(LOPNFRAME,1,3,I,1);
    ASSIGNTEMP(T);
    CODE[1]:=MOVCODE OR BOOL(FLDSIZE*ROPNFRAME[4])
                    OR BOOL(T);
    INSERTCODE(CODE,LOPNFRAME[2],1);
    UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],1);
  END;
  IF OPRTYPE[OP] AND ALLOCTEMPOUT THEN
            ASSIGNTEMP(TEMPOUT)
  ELSE TEMPOUT:=ROPNFRAME[4];
  MAKECODE(CODE,LOPNFRAME,ROPNFRAME,OP,TEMPOUT);
  INSERTCODE(CODE,LOPNFRAME[2],NINSTRS[OP]);
  UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],NINSTRS[OP]);
  FOR I:=1 UNTIL 7+LOPNFRAME[7] DO
            MFRAME[I]:=LOPNFRAME[I];
  MFRAME[4]:=TEMPOUT;
  PUSHFRAME(STACK,MFRAME);
END
ELSE
BEGIN
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    RI:=DEFLFRAME(ROPNFRAME);
```



```
  ASSIGNTEMP(RT);
  CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(RT);
  I:=MAKESPLADR(RI,ORANK[RI]+1,1);
  INSERTCODE(CODE,I,1);
  ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=I+1;
  ROPNFRAME[4]:=RT;
END;
IF OP#ASSGNOP OR (BOOL(LOPNFRAME[1]) AND LMASK) THEN
BEGIN
  IF BOOL(LOPNFRAME[1]) AND LMASK THEN
  BEGIN
    K:=INT(BOOL(LOPNFRAME[1]) AND SYMBMASK);
    ASSIGNLNUM(LI,K);
    IF OP=ASSGNOP THEN
    BEGIN
      FOR I:=6 UNTIL 7+ROPNFRAME[7] DO
      BEGIN
        IF ROPNFRAME[I]   -1 THEN
        BEGIN INTEGER T;
          T:= (-ROPNFRAME[I]) REM 64;
          ROPNFRAME[I]:=(-ROPNFRAME[I]) DIV 64;
          IF I=6 THEN
            FIXLIST[(FLP:=FLP+1)]:=MAKEFIXLIST(LI,5,T)
          ELSE IF I=7 THEN FIXLIST[(FLP:=FLP+1)]:=
                  MAKEFIXLIST(LI,4,T)
          ELSE FIXLIST[(FLP:=FLP+1)]:=
                  MAKEFIXLIST(LI,9+(I-8)*4,T);
        END;
        LOPNFRAME[I]:=ROPNFRAME[I];
      END;
      ORANK[LI]:=ROPNFRAME[7];
      FOR I:=1 UNTIL ORANK[LI] DO
      BEGIN
        ORHO[LI,I]:=ROPNFRAME[I+7];  ODELTA[LI,I]:=1;
      END;
    END
    ELSE
    BEGIN
      LOPNFRAME[6]:=INT(OPNTYPE[K]);
      LOPNFRAME[7]:=ORANK[LI];
      FOR I:=1 UNTIL ORANK[LI] DO
        LOPNFRAME[I+7]:=RHO[K,I];
    END;
    LRNK:=ORANK[LI];
    GENSCODE(LI,LRNK);
    LOPNFRAME[1]:=MAKESPLADR(LI,LRNK+1,1);
    LOPNFRAME[2]:=LOPNFRAME[1];
    LOPNFRAME[3]:=LOPNFRAME[1];
    LOPNFRAME[5]:=LI;
    LOPNFRAME[4]:=0;
  END;
  IF CONFORMCHK(LOPNFRAME,ROPNFRAME,OP) THEN
  BEGIN
```

```
        ASSIGNLREG(LR);
        LNKREG[LR]:=LOPNFRAME[5];
        CODE[1]:=CRJCODE OR BOOL(LR);
        INSERTCODE(CODE,ROPNFRAME[1],1);
        UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[1],1);
        UPDATEPOINTER(LOPNFRAME,1,3,ROPNFRAME[1],1);
        TEMPOUT:=ROPNFRAME[4];
        IF OPRTYPE[OP] AND ALLOCTEMPOUT THEN
           ASSIGNTEMP(TEMPOUT);
        MAKECODE(CODE,LOPNFRAME,ROPNFRAME,OP,TEMPOUT);
        INSERTCODE(CODE,LOPNFRAME[2],NINSTRS[OP]);
        UPDATEPOINTER(ROPNFRAME,1,3,LOPNFRAME[2],
                                        NINSTRS[OP]);
        UPDATEPOINTER(LOPNFRAME,1,3,LOPNFRAME[2],
                                        NINSTRS[OP]);
        CODE[1]:=CRJCODE OR BOOL(LR);
        INSERTCODE(CODE,LOPNFRAME[3],1);
        MERGEFRAME(MFRAME,LOPNFRAME,ROPNFRAME,TEMPOUT);
        PUSHFRAME(STACK,MFRAME);
      END
      ELSE
        WRITE("[N]NON CNFRMBL ARRAYS FOR DSCAL OP")
    END
    ELSE
     WRITE("[N]L OP  OF ASSIGN OP NOT SIMPLE LADDER");
  END;
END;



PROCEDURE MONADSCAL(STACK,OP);  INTEGER OP;  VALUE OP;
            INTEGER ARRAY STACK;

BEGIN     INTEGER RI,RT,T,RRNK,I,K;
    INTEGER ARRAY ROPNFRAME,MFRAME[1:STACK[0]];
    BOOLEAN ARRAY CODE[1:10];
  POPFRAME(STACK,ROPNFRAME);
  IF BOOL(ROPNFRAME[1]) AND LMASK THEN
  BEGIN
    RI:=DEFLFRAME(ROPNFRAME);
    I:=MAKESPLADR(RI,ORANK[RI]+1,1);
    ASSIGNTEMP(T);
    CODE[1]:=MOVCODE OR ATPISOURCE OR BOOL(T);
    INSERTCODE(CODE,I,1);
    ROPNFRAME[1]:=ROPNFRAME[2]:=ROPNFRAME[3]:=I+1;
    ROPNFRAME[4]:=T;
  END;
  RT:=ROPNFRAME[4];
  IF OPRTYPE[OP] AND ALLOCTEMPOUT THEN ASSIGNTEMP(RT);
  MAKECODE(CODE,ROPNFRAME,ROPNFRAME,OP,RT);
  INSERTCODE(CODE,ROPNFRAME[2],NINSTRS[OP]);
```



```
  UPDATEPOINTER(ROPNFRAME,1,3,ROPNFRAME[2],NINSTRS[OP]);
  ROPNFRAME[4]:=RT;
  PUSHFRAME(STACK,ROPNFRAME);
END;



PROCEDURE OPRAND(STACK,SYMB);  BOOLEAN SYMB; VALUE SYMB;
    INTEGER ARRAY STACK;

BEGIN   INTEGER I;  INTEGER ARRAY SFRAME[1:STACK[0]];
  SFRAME[1]:=INT(SYMB OR LMASK);
  FOR I:= 2 UNTIL STACK[0] DO SFRAME[I]:=0;
  PUSHFRAME(STACK,SFRAME);
END;



PROCEDURE OPRATR(STACK,SYMB); BOOLEAN SYMB; VALUE SYMB;
                    INTEGER ARRAY STACK;

BEGIN   INTEGER OP,OPTYPE;
  OP:=INT(SYMB AND SYMBMASK);   OPTYPE:=INT(OPRTYPE[OP]);
  IF OPRTYPE[OP] AND DYADSCALMASK THEN
            DYADSCAL(STACK,OP)
  ELSE IF OPRTYPE[OP] AND MONADSCALMASK THEN
            MONADSCAL(STACK,OP)
  ELSE IF OPTYPE=INT(OUTPRODINSTR) THEN
            OUTPROD(STACK)
  ELSE IF OPTYPE=INT(REDUCINSTR) THEN
             REDUC(STACK)
  ELSE IF OPTYPE=INT(IPINSTR) THEN
            INNERPROD(STACK)
  ELSE IF OPTYPE=INT(SCANINSTR) THEN
            SCAN(STACK)
  ELSE IF OPTYPE=INT(MBRSHPINSTR) THEN
            MEMBERSHIP(STACK)
  ELSE IF OPTYPE=INT(INDXGENINSTR) THEN
            INDEXGEN(STACK)
  ELSE IF OPTYPE=INT(INDXOFINSTR) THEN
            INDEXOF(STACK)
  ELSE IF OPTYPE=INT(REVERSEINSTR) THEN
            REVERSE(STACK)
  ELSE IF OPTYPE=INT(SHAPEINSTR) THEN
            SHAPE(STACK)
  ELSE IF OPTYPE=INT(RSTRCTRINSTR) THEN
            RESTRUCTURE(STACK)
  ELSE IF OPTYPE=INT(TAKEINSTR) THEN TAKE(STACK)
  ELSE IF OPTYPE=INT(DROPINSTR) THEN DROP(STACK)
  ELSE IF OPTYPE=INT(TRMINSTR) THEN MONADTRANSPOSE(STACK)
  ELSE IF OPTYPE=INT(TRDINSTR) THEN DYADTRANSPOSE(STACK)
```

```
   ELSE IF OPTYPE=INT(RAVELCODE) THEN RAVEL(STACK)
   ELSE IF OPTYPE=INT(CATENATECODE) THEN CATENATE(STACK)
   ELSE IF OPTYPE=INT(COMPRESSCODE) THEN COMPRESS(STACK)
   ELSE IF OPTYPE=INT(EXPANDCODE) THEN EXPAND(STACK)
   ELSE IF OPTYPE=INT(DECODEINSTR) THEN DECODE(STACK)
   ELSE IF OPTYPE=INT(ENCODEINSTR) THEN ENCODE(STACK)
   ELSE WRITE("[N]OPERATOR NOT HANDLED AT THIS TIME");
   PRINTSTACK(STACK);
END;



PROCEDURE OUTSCODE(L);     INTEGER L;     VALUE L;

BEGIN  INTEGER J,K;     BOOLEAN INSTR;
  FOR J:=1 UNTIL 1+2*ORANK[L] DO
  BEGIN
    FOR K:=1 UNTIL INT(SCODE[L,J,0]) DO
    BEGIN
      IF MATCH(SCODE[L,J,K],ENDMARK) THEN
      BEGIN
        INSTR:=BRCODE OR BOOL(INT(SCODE[L,J,0])-K);
        PRINTOCTAL(INSTR);
      END
      ELSE PRINTOCTAL(SCODE[L,J,K]);
      NEWLINE;
    END;
    PRINTOCTAL(ENDMARK);    NEWLINE;
  END;
  NEWLINE(2);
END;



PROCEDURE OUTLNKREG;

BEGIN  INTEGER I;
  PRINT(NXTLAD-2,4);  SPACE(4);
  FOR I:=1 UNTIL NXTLAD-2 DO PRINT(LNKREG[I],4);
  NEWLINE;
END;



PROCEDURE OUTTINIT;

BEGIN  INTEGER I;
  I:=1;
  WHILE TINITLIST[I]#0 DO
  BEGIN  NEWLINE;
    PRINT(TINITLIST[I],4); SPACE(4);
    PRINT(TINITLIST[I+1],4);
```



```
    I:=I+2;
  END;
  NEWLINE; PRINT(0,4);  NEWLINE;
END;



PROCEDURE OUTFIXLIST;
BEGIN
      INTEGER I;

  FOR I:=1 UNTIL FLP DO
  BEGIN
    PRINTOCTAL(FIXLIST[I]);     SPACE(4);
    IF (I REM 4) = 0 THEN NEWLINE;
  END;
  NEWLINE;     PRINTOCTAL(0);    NEWLINE;
END;



PROCEDURE OUTMEM;
BEGIN          INTEGER I;

  OPENFILE(4,"NMEM.DAT");
  PRINT(NWORDS);       NEWLINE;
  FOR I:=1 UNTIL NWORDS DO
  BEGIN
    PRINT(DATA[I],5);
    IF I REM 15 = 0 THEN NEWLINE;
  END;
  NEWLINE;
  CLOSEFILE(4);
END;



PROCEDURE OUTDAT;

BEGIN    INTEGER I,J;
  OUTPUT(4,"DSK");  OPENFILE(4,OUTFILE);  SELECTOUTPUT(4);
  PRINT(NXTLAD-1,4);
  FOR I:=1 UNTIL NXTLAD-1 DO
  BEGIN
    NEWLINE(2);
    PRINT(ORANK[I],4);  SPACE(4);  PRINT(OBETA[I],6);
    PRINT(OOPNTYPE[I],4);  NEWLINE;
```

```
      FOR J:=1 UNTIL ORANK[I] DO
      BEGIN
        PRINT(ORHO[I,J],4); PRINT(ODELTA[I,J],4);  NEWLINE;
      END;
      NEWLINE;
      OUTSCODE(I);
    END;
    OUTLNKREG;  OUTTINIT;  OUTFIXLIST;
    PRINT(STARTLAD,4); NEWLINE;  !STARTING LADDER NUMBER;
    CLOSEFILE(4);
    OUTMEM;
    RELEASE(4);          SELECTOUTPUT(-1);
  END;



  PROCEDURE INSERTHALT(STACK);  INTEGER ARRAY STACK;
  BEGIN

        INTEGER RNK,N,K;
        INTEGER ARRAY LFRAME[1:STACK[0]];
        BOOLEAN ARRAY CODE[1:1];
    POPFRAME(STACK,LFRAME);
    STARTLAD:=LFRAME[5];
    RNK:=ORANK[STARTLAD];
    N:=INT(SCODE[STARTLAD,2*RNK+1,0]);
    K:=MAKESPLADR(STARTLAD,2*RNK+1,N);
    CODE[1]:=HALTCODE;
    INSERTCODE(CODE,K,1);
  END;



COMMENT THIS IS THE MAIN PROGRAM;

  BEGIN   BOOLEAN SYMB;
    SETUP;
    OUTPUT(4,"DSK"); OPENFILE(4,"TEMP.DAT");
    SELECTOUTPUT(4);
    PRNTOPRS;    PRNTOPNS;
    INPUT(3,"DSK");  OPENFILE(3,INFILE);
    SELECTINPUT(3);
    WHILE NEXTSYMB(SYMB,3) DO
    BEGIN
      PRINTOCTAL(SYMB);  NEWLINE;
      IF SYMB AND OPNMASK THEN OPRAND(OPNDSTK,SYMB)
      ELSE OPRATR(OPNDSTK,SYMB);
    END;
    IF OPNDSTK[0] = OPNDSTK[-2] THEN
```



```
      BEGIN
        INSERTHALT(OPNDSTK);
        OUTDAT;
      END
      ELSE WRITE("[N]BAD TERMINAL STACK");
    END;
  END;
```

APPENDIX F

SIMULATOR FOR THE LADDER PROCESSOR

This appendix gives an ALGOL program that simulates the high-speed ladder processor (Appendix C). The program reads the splice code from a file prepared by the program descirbed in Appendix E. This program generates the ladder code and completes the construction of the ladder network.

The program counts instructions as it executes the network. This instruction count is used to estimate the time required for a real high-speed ladder processor to execute the network. The program also computes the distribution of the number of ladder processor instructions executed between references to the host computer's memory. With this distribution it is possible to assess the impact of each machine on the other.

A few of the instructions listed in Appendix C are not implemented; these are the dyadic circle function, the inverse hyperbolic functions, and roll. The program uses only the integer data type.

The simulation program, running on a KA-10 processor, requires a factor of 30,000 more time that the processor it simulates to execute a ladder network. The high-speed ladder processor executes an instruction about twenty times faster than the KA-10 processor so the program executes roughly 1500 PDP-10 instructions for each instruction

simulated.

The two programs can't communicate as freely as the machines they model so that the tasks are not divided between the two programs as they would be between the two machines. For example, this program reads in the splice code and generates the ladder code. But the ladder processor generates no code; it is completely loaded by the host machine.

```
COMMENT                    THIS IS FILE FL.ALG;



BEGIN
  CHECKON 1;
  COMMENT  DECLARE THE GLOBAL VARIABLES;

  BOOLEAN ARRAY CODE[0:1000],OPNTYPE[1:20];
  INTEGER ARRAY LADID[1:9],LDATA[1:9,1:40],
                DATA[1:1000],GDATA[1:256],CRJREG[1:10],
                RANK[1:20],RHO[1:20,1:10],FIXLIST[1:10],
                IBMMREF,IBS,IBD[0:15];
  STRING ARRAY OPNNAME[1:20];
  BOOLEAN ENDSPLCODE,INITPICODE,MOVORIGCODE,ISTEPCODE,
          ISTEPOFFSETMASK,
          BRCODE,BROFFSETMASK,PCMASK,ISTEPMASK,CRJMASK,
          BADCODE,ZCC,GCC,NCC,LSB,PARAMMASK,
          DOPMASK,SOPMASK,HALT,
          RUN,TRACE,READFLAG,ENDMARK;
  INTEGER NLADRS,NSCADRB,FIELDSIZE,ADRFIELD,DNI,SOPCODESIZE,
          NINSTRS,NCRJREGMAX,STARTLAD,NOPNS,ISTEPOFFSETSIZE,
          NATPISOURCE,NATPIDEST,MAXHISTBINS,DNIS,DNID;

COMMENT THIS BOOLEAN PROCEDURE CHECKS TO SEE IF TWO BOOLEAN
VARIABLES ARE EXACTLY EQUAL;

  BOOLEAN PROCEDURE MATCH(A,B); BOOLEAN A,B; VALUE A,B;
  BEGIN
    MATCH:=IF INT(A EQV B) = -1 THEN TRUE ELSE FALSE;
  END;




  BOOLEAN PROCEDURE TF(X);  BOOLEAN X; VALUE X;
  BEGIN
    TF:= IF X THEN TRUE ELSE FALSE;
  END;


COMMENT THIS PROCEDURE PRINTS OUT THE DATA FOR LADDER I ;

  PROCEDURE DUMPLDAT(I); INTEGER I; VALUE I;
  BEGIN INTEGER J;
    NEWLINE;
    WRITE("LADDER DATA FOR LADDER # "); PRINT(I,4);NEWLINE;
    FOR J:=1 STEP 4 UNTIL 8+4*(LDATA[I,4]) DO
      BEGIN
```



```
        PRINT(LDATA[I,J],5);
        PRINT(LDATA[I,J+1],5);
        PRINT(LDATA[I,J+2],5);
        PRINT(LDATA[I,J+3],5);
        NEWLINE;
      END;
  END;



COMMENT THIS PROCEDURE PRINTS OUT AN ENTIRE ROW OF THE DATA
        ASSOCIATED WITH LADDER LNUM WITH UP TO SIXTEEN
        ELEMENTS PER LINE.  THAT IS IT SWEEPS THE
        RIGHT MOST SUBSCRIPT THROUGH ITS ENTIRE RANGE AND
        PRINTS OUT EACH ARRAY ELEMENT AS IT GOES.
        IT ACCESSES THE DATA IN EXACTLY THE SAME WAY THE
        LADDER DOES AND IT TAKES CARE OF OF MULTI LINE ROWS.
        PI IS THE DATA POINTER AND N IS THE NUMBER OF
        ELEMENTS PER ROW.;


  PROCEDURE PRNTLINE(LNUM,RANK,PI,N);
       INTEGER LNUM,RANK,PI,N;
       VALUE LNUM,RANK,N;
  BEGIN  INTEGER I,DELTAR;
    DELTAR:=LDATA[LNUM,9+(RANK-1)*4];
    FOR I:=1 UNTIL N DO
    BEGIN
      IF I 16 AND (I REM 16) = 1 THEN NEWLINE;
      PRINT(DATA[PI],4);
      PI:=PI+DELTAR;                    !PI:=PI+DELTA[RANK];
    END;
    PI:=PI-DELTAR ;                     !PI:=PI-DELTA[RANK];
  END;

COMMENT THIS PROCEDURE PRINTS OUT THE HEADER INFORMATION
        FOR LADDER LNUM -- BETA (BASE ADDRESS), RANK,
        RHO, DELTA, AND G;

  PROCEDURE PRNTHDR(LNUM); VALUE LNUM; INTEGER LNUM;
  BEGIN INTEGER BETA,RANK,I;
    BETA:=LDATA[LNUM,3];
    RANK:=LDATA[LNUM,4];
    NEWLINE; WRITE("DATA FOR LADDER NUMBER"); PRINT(LNUM,4);
    NEWLINE;
    WRITE("BASE ADDRESS IN SHARED STORAGE"); PRINT(BETA,5);
    NEWLINE;
    WRITE("RANK "); PRINT(RANK,4); NEWLINE;
    WRITE("SHAPE");
    FOR I:=0 UNTIL RANK-1 DO
        PRINT(LDATA[LNUM,10+I*4],5);  !RHO;
```

```
    NEWLINE;
    WRITE("DELTA");
    FOR I:=0 UNTIL RANK-1 DO
        PRINT(LDATA[LNUM,9+I*4],5);  !DELTA;
    NEWLINE;
    WRITE("  G  ");
    FOR I:=0 UNTIL RANK-1 DO
        PRINT(LDATA[LNUM,11+I*4],5);  !G;
    NEWLINE(2);
    WRITE("                DATA"); NEWLINE(2);
  END;

COMMENT  THIS PROCEDURE OUTPUTS LADDER PARAMETERS AND DATA
         IN A CONVENIENT FORMAT;

  PROCEDURE PRNTLAND(LNUM); VALUE LNUM; INTEGER LNUM;
  BEGIN   INTEGER PROD,TPROD,RANK,BETA,PI,I,J,K,L;
    BETA:=LDATA[LNUM,3];
    RANK:=LDATA[LNUM,4];
    PRNTHDR(LNUM);
    IF BETA#0 THEN
    BEGIN
      PROD:=1;
      FOR I:=0 UNTIL RANK-1 DO
              PROD:=PROD*LDATA[LNUM,10+I*4];
      K:=LDATA[LNUM,10+4*(RANK-1)];
      J:=PROD DIV K;
      PI:=LDATA[LNUM,3];

COMMENT PROD IS THE NUMBER OF DATA ELEMENTS IN THE LADDER
  K IS THE MAX VALUE OF THE MOST RAPIDLY VARYING SUBSCRIPT
  J IS THE NUMBER OF TIMES A VECTOR OF LENGTH K OCCURS IN
    THE ARRAY;

      FOR I:=1 STEP 1 UNTIL J DO
      BEGIN
        PRNTLINE(LNUM,RANK,PI,K);
        PROD:=PROD-K;
        IF K 16 THEN NEWLINE;
        TPROD:=1;
        FOR L:=RANK   STEP -1 UNTIL 2 DO
        BEGIN
          TPROD:=TPROD*LDATA[LNUM,10+(L-1)*4];
          IF (PROD REM TPROD) = 0 THEN
          BEGIN
            IF L#RANK AND(PROD REM
                  (TPROD DIV LDATA[LNUM,10+(L-1)*4])=0)
              THEN PI:=PI-LDATA[LNUM,9+(L-1)*4];
            NEWLINE;
            PI:=PI+LDATA[LNUM,9+(L-2)*4];
```



```
          END;
        END;
      END;
    END
    ELSE WRITE(" DATA GENERATED AT RUN TIME NOT STORED[N]");
  END;


COMMENT  PROCEDURE TO READ IN OR INITIALIZE LADDER DATA;

  PROCEDURE GETLDAT(I); INTEGER I;  VALUE I;
  BEGIN     INTEGER RANK,J;
    READ(RANK);
    LDATA[I,6]:=1;                !ORIGIN;
    READ(LDATA[I,3]);             !BETA;
    LDATA[I,4]:=RANK;
    READ(LDATA[I,5]);             !TYPE;
    FOR J:=0 UNTIL RANK-1 DO
    BEGIN
      READ(LDATA[I,J*4+10]);      !RHO;
      READ(LDATA[I,J*4+9]);       !DELTA;
    END;
  END;

COMMENT  PROCEDURE TO READ IN THE SPLICE CODE;

 PROCEDURE FILLSPLICE;
  BEGIN
    BOOLEAN INSTR;
    READOCTAL(INSTR);
    WHILE NOT MATCH(INSTR,ENDSPLCODE) DO
    BEGIN
      CODE[NINSTRS]:=INSTR;
      READOCTAL(INSTR);
      NINSTRS:=NINSTRS+1;
    END
  END;


  PROCEDURE GETSCODE(LNUM);     INTEGER LNUM;  VALUE LNUM;
  BEGIN         INTEGER I,RANK;
                 INTEGER ARRAY DEST[0:10];
    LDATA[LNUM,2]:=NINSTRS;              !SETUP LADDERR PC;
    RANK:=LDATA[LNUM,4];
    DEST[0]:=NINSTRS;
    IF RANK=0 THEN
    BEGIN
      CODE[NINSTRS]:=INITPICODE;
```

```
      NINSTRS:=NINSTRS+1;
      FILLSPLICE;
    END
    ELSE
    BEGIN
      FILLSPLICE;
      CODE[NINSTRS]:=INITPICODE;
      NINSTRS:=NINSTRS+1;
      FOR I:=1 UNTIL RANK DO
      BEGIN
        CODE[NINSTRS]:=MOVORIGCODE OR BOOL(4*(I+1));
        DESTXI]:=NINSTRS:=NINSTRS+1;
        FILLSPLICE;
      END;
      FOR I:=RANK STEP -1 UNTIL 1 DO
      BEGIN
        CODE[NINSTRS]:=ISTEPCODE OR BOOL((I+1)*
                       ISTEPOFFSETSIZE) OR
                       ISTEPOFFSETMASK AND
                       BOOL(DEST[I]-NINSTRS-1);
        NINSTRS:=NINSTRS+1;
        FILLSPLICE;
      END;
    END;
    CODE[NINSTRS]:=BRCODE OR (BROFFSETMASK AND
                    BOOL(DEST[0]-NINSTRS-1));
    NINSTRS:=NINSTRS+1;
  END;



COMMENT  PROCEDURE TO COMPUTE THE DELTAS FROM THE G'S;

  PROCEDURE COMPDELTA(LNUM); VALUE LNUM; INTEGER LNUM;
  BEGIN  INTEGER RANK,I,J;
    RANK:=LDATA[LNUM,4];
    LDATA[LNUM,RANK*4+5]:=LDATA[LNUM,RANK*4+7];
    FOR I:=RANK-1 STEP -1 UNTIL 1 DO
      LDATA[LNUM,9+(I-1)*4]:=LDATA[LNUM,11+(I-1)*4] +
        LDATA[LNUM,I*4+9] - LDATA[LNUM,I*4+10]*
                LDATA[LNUM,I*4+11];
  END;


COMMENT  PROCEDURE TO COMPUTE THE G'S FROM THE DELTAS;


  PROCEDURE COMPG(LNUM); VALUE LNUM; INTEGER LNUM;
  BEGIN  INTEGER RANK,I,J;
    RANK:=LDATA[LNUM,4];
    LDATA[LNUM,RANK*4+7]:=LDATA[LNUM,RANK*4+5];
    FOR I:=RANK-1 STEP -1 UNTIL 1 DO
```



```
    LDATA[LNUM,11+(I-1)*4]:=LDATA[LNUM,9+(I-1)*4] +
      LDATA[LNUM,I*4+10]*LDATA[LNUM,I*4+11]-
              LDATA[LNUM,I*4+9];
END;




PROCEDURE GETDAT;
BEGIN   INTEGER I,NWORDS;
  INPUT(3,"DSK");  OPENFILE(3,"NMEM.DAT");
  SELECTINPUT(3);
  READ(NWORDS);
  FOR I:=1 UNTIL NWORDS DO READ(DATA[I]);
  CLOSEFILE(3);
END;


PROCEDURE GETCRJREG;
BEGIN INTEGER NREG,I;
  READ(NREG);
  IF NREG   NCRJREGMAX THEN
        WRITE("TOO MANY COROUTINE JUMP REGISTERS")
    ELSE FOR I:=1 UNTIL NREG DO READ(CRJREG[I]);
 END;


PROCEDURE GETTINIT;
BEGIN INTEGER I;
  I:=1;
  WHILE I#0 DO
  BEGIN
    READ(I);
    IF I#0 THEN READ(GDATA[I]);
  END;
END;



PROCEDURE GETFIXLIST;
BEGIN
      INTEGER I;

  I:=1;
  READOCTAL(FIXLIST[I]);
  WHILE FIXLIST[I] # 0 DO READOCTAL(FIXLIST[(I:=I+1)]);
END;



PROCEDURE SETUP;
```

```
BEGIN
  INTEGER I,K;

  NCRJREGMAX:=10;
  NSCADRB:=8;                       FIELDSIZE:=2**NSCADRB;
  SOPMASK:=BOOL(2 14);              BRCODE:=BOOL(2 15);
  DOPMASK:=BOOL(63*FIELDSIZE 2);    GCC:=FALSE;
  BROFFSETMASK:=PCMASK:=%7777;      ISTEPCODE:=%0;
  ISTEPOFFSETSIZE:=1024;            ENDSPLCODE:=BOOL(-1);
  ISTEPOFFSETMASK:=%1777;           ISTEPMASK:=%16000;
  ZCC:=FALSE;                       CRJMASK:=%100;
  MAXHISTBINS:=15;                  MOVORIGCODE:=%6603000;
  NINSTRS:=0;                       INITPICODE:=%6601401;
  NATPISOURCE:=0;                   NATPIDEST:=0;
  SOPCODESIZE:=64;                  PARAMMASK:=%340;
  DNIS:=0;
  DNID:=0;
  DNI:=0;

  FOR I:=0 UNTIL MAXHISTBINS DO IBS[I]:=
                      IBD[I]:=IBMMREF[I]:=0;

  TRACE:=FALSE;
  RUN := TRUE;


  INPUT(3,"DSK"); OPENFILE(3,"CODE.DAT");
  SELECTINPUT(3);
  IF TRACE THEN
    BEGIN
      OUTPUT(4,"DSK");OPENFILE(4,"OUT.LST");
      SELECTOUTPUT(4);
      WRITE("LADDER #      PC           INSTR  ");
      NEWLINE(2);
    END;
  READ (NLADRS);
  FOR I:=1 UNTIL NLADRS DO
  BEGIN
    GETLDAT(I);
    GETSCODE(I);
    COMPG(I);
    DUMPLDAT(I);
  END;
  GETCRJREG;
  GETTINIT;
  GETFIXLIST;
  READ(STARTLAD);
  CLOSEFILE(3);
  GETDAT;
  RELEASE(3);
  NINSTRS:=0;
END;
```



```
PROCEDURE PRNTDATA(DAT); INTEGER DAT; VALUE DAT;
BEGIN
  OWN INTEGER NCOUNT;
  IF NCOUNT REM 4 = 0 THEN NEWLINE;
  NCOUNT:=NCOUNT + 1;
  PRINT(DAT,6);
END;

INTEGER PROCEDURE CIR(A,B);  INTEGER A,B; VALUE A,B;
BEGIN
  WRITE("DYADIC CIRCLE FUNCTIION NOT INSTALLED");
  NEWLINE;    CIR:=0;
END;



INTEGER PROCEDURE FACT(A);  INTEGER A;  VALUE A;
BEGIN  INTEGER I,T;
  T:=1;
  FOR I:=2 UNTIL A DO T:=T*I;
  FACT:=T;
END;


INTEGER PROCEDURE COMB(A,B); INTEGER A,B;  VALUE A,B;
BEGIN
  COMB:=FACT(B) DIV (FACT(A)*FACT(B-A));
END;


INTEGER PROCEDURE ARCSINH(A);    INTEGER A;     VALUE A;
BEGIN
  WRITE("ARCSINH NOT INSTALLED");
  NEWLINE;    ARCSINH:=0;
END;

INTEGER PROCEDURE ARCCOSH(A);    INTEGER A;     VALUE A;
BEGIN
  WRITE("ARCCOSH NOT INSTALLED");
  NEWLINE;    ARCCOSH:=0;
END;

INTEGER PROCEDURE ARCTANH(A);    INTEGER A;     VALUE A;
BEGIN
  WRITE("ARCTANH NOT INSTALLED");
  NEWLINE;      ARCTANH:=0;
END;


INTEGER PROCEDURE ROLL(A);  INTEGER A;   VALUE A;
BEGIN
```

```
  WRITE("ROLL NOT INSTALLED");  NEWLINE;  ROLL:=0;
END;


INTEGER PROCEDURE CEL(A);  INTEGER A;   VALUE A;
BEGIN
  CEL:=A;
END;


INTEGER PROCEDURE FLR(A);  INTEGER A;   VALUE A;
BEGIN
  FLR:=A;
END;


PROCEDURE ISTEP(LNUM,INSTR);  INTEGER LNUM;
             BOOLEAN INSTR;
             VALUE LNUM,INSTR;
BEGIN        INTEGER SUBNUM,RHO,I,DELTA,OFFSET,DFIELD,K;
  SUBNUM:=((INT(INSTR) DIV ISTEPOFFSETSIZE) REM 8)-1;
  RHO:=LDATA[LNUM,SUBNUM*4+6];
  I:= LDATA[LNUM,SUBNUM*4+4];
  DELTA:=LDATA[LNUM,SUBNUM*4+5];
  IF TRACE THEN
  BEGIN
    SPACE(4); WRITE("RHO,I,DELTA  ");
    PRINT(RHO,4);PRINT(I,4);
    PRINT(DELTA,4);NEWLINE;
  END;

  IF I RHO THEN
  BEGIN
    LDATA[LNUM,SUBNUM*4+4]:=I+1;
    LDATA[LNUM,1]:=LDATA[LNUM,1]+DELTA;
    DFIELD:=INT(INSTR AND ISTEPOFFSETMASK);
    K:=ISTEPOFFSETSIZE;
    OFFSET:=DFIELD - (IF DFIELD =K DIV 2 THEN K ELSE 0);
    IF LDATA[LNUM,2]+OFFSET  = 0 AND
              LDATA[LNUM,2]+OFFSET K*4 THEN
         LDATA[LNUM,2]:=LDATA[LNUM,2]+OFFSET
       ELSE
       BEGIN  NEWLINE;
         WRITE ("BRANCH OUT OF BOUNDS");
         PRINT (OFFSET,5);
         PRINT(DFIELD,5); PRINT(LDATA[LNUM,2],6);
       END;
  END;
END;


PROCEDURE CRJMP(LNUM,INSTR);
```



```
INTEGER LNUM;   BOOLEAN INSTR;    VALUE INSTR;
BEGIN  INTEGER T,REGNUM;
  T:=LNUM;    REGNUM:=INT(INSTR AND %77);
  IF TRACE THEN
  BEGIN
    SPACE(4); WRITE("REGNUM,NEXTLAD  "); PRINT(REGNUM,4);
    PRINT(CRJREG[REGNUM],4);  NEWLINE;
  END;
  IF REGNUM   NCRJREGMAX THEN
      WRITE("LINK REGISTER OUT OF BOUNDS")
  ELSE
  BEGIN
    LNUM:=CRJREG[REGNUM];
    CRJREG[REGNUM]:=T;
  END;
END;



PROCEDURE BRANCH(LNUM,DFIELD);
INTEGER LNUM,DFIELD; VALUE LNUM,DFIELD;
BEGIN  INTEGER OFFSET,OSIZE;
  OSIZE:=INT(BROFFSETMASK)+1;
  OFFSET:=DFIELD -(IF DFIELD =OSIZE DIV 2 THEN
              OSIZE ELSE 0);
  IF LDATA[LNUM,2]+OFFSET  = 0 AND
              LDATA[LNUM,2]+OFFSET OSIZE THEN
       LDATA[LNUM,2]:=LDATA[LNUM,2]+OFFSET
     ELSE
     BEGIN  NEWLINE;
       WRITE ("BRANCH OUT OF BOUNDS"); PRINT (OFFSET,5);
       PRINT(DFIELD,5); PRINT(LDATA[LNUM,2],6);
     END;
END;

PROCEDURE SXQT(ICODE,DEST);
INTEGER ICODE,DEST;
VALUE ICODE;
BEGIN          REAL X;
  IF ICODE=1 THEN DEST:=DEST+1
  ELSE IF ICODE=2 THEN DEST:=DEST-1
  ELSE IF ICODE=3 THEN
    BEGIN
      ZCC:=DEST=0;
      NCC:=DEST 0;
      GCC:=FALSE;
      LSB:=TF(BOOL(DEST) AND %1);
    END
  ELSE IF ICODE=4 THEN DEST:=0
  ELSE IF ICODE=5 THEN DEST:=-DEST
  ELSE IF ICODE=6 THEN DEST:=INT(NOT BOOL(DEST))
  ELSE IF ICODE=7 THEN DEST:=1/DEST
  ELSE IF ICODE=8 THEN DEST:=SIGN(DEST)
```

```
  ELSE IF ICODE=9 THEN DEST:=CEL(DEST)
  ELSE IF ICODE=10 THEN DEST:=FLR(DEST)
  ELSE IF ICODE=11 THEN DEST:=ABS(DEST)
  ELSE IF ICODE=12 THEN DEST:=ROLL(DEST)
  ELSE IF ICODE=13 THEN DEST:=EXP((X:=DEST))
  ELSE IF ICODE=14 THEN DEST:=LN((X:=DEST))
  ELSE IF ICODE=15 THEN DEST:=FACT(DEST)
  ELSE IF ICODE=16 THEN DEST:=3.14159265*DEST
  ELSE IF ICODE=17 THEN DEST:=SIN((X:=DEST))
  ELSE IF ICODE=18 THEN DEST:=COS((X:=DEST))
  ELSE IF ICODE=19 THEN DEST:=TAN((X:=DEST))
  ELSE IF ICODE=20 THEN DEST:=(1+DEST 2) .5
  ELSE IF ICODE=21 THEN DEST:=SINH((X:=DEST))
  ELSE IF ICODE=22 THEN DEST:=COSH((X:=DEST))
  ELSE IF ICODE=23 THEN DEST:=TANH((X:=DEST))
  ELSE IF ICODE=24 THEN DEST:=(1-DEST 2) .5
  ELSE IF ICODE=25 THEN DEST:=ARCSIN((X:=DEST))
  ELSE IF ICODE=26 THEN DEST:=ARCCOS((X:=DEST))
  ELSE IF ICODE=27 THEN DEST:=ARCTAN((X:=DEST))
  ELSE IF ICODE=28 THEN DEST:=((-1)+DEST 2) .5
  ELSE IF ICODE=29 THEN DEST:=ARCSINH((X:=DEST))
  ELSE IF ICODE=30 THEN DEST:=ARCCOSH((X:=DEST))
  ELSE IF ICODE=31 THEN DEST:=ARCTANH((X:=DEST));
  IF TRACE THEN
  BEGIN
    PRINT(DEST,5); NEWLINE;
  END;
END;


PROCEDURE DXQT(ICODE,SOURCE,DEST);
    INTEGER ICODE,SOURCE,DEST;
    VALUE SOURCE,ICODE;
BEGIN         REAL X,Y;
  IF ICODE=1 THEN DEST:=-INT(SOURCE DEST)
  ELSE IF ICODE=2 THEN DEST:=-INT(SOURCE =DEST)
  ELSE IF ICODE=3 THEN DEST:=-INT(DEST=SOURCE)
  ELSE IF ICODE=4 THEN DEST:=-INT(SOURCE =DEST)
  ELSE IF ICODE=5 THEN DEST:=-INT(SOURCE DEST)
  ELSE IF ICODE=6 THEN DEST:=-INT(SOURCE#DEST)
  ELSE IF ICODE=7 THEN
            DEST:=INT(BOOL(SOURCE) OR BOOL(DEST))
  ELSE IF ICODE=8 THEN
            DEST:=INT(BOOL(DEST) AND BOOL(SOURCE))
  ELSE IF ICODE=9 THEN
            DEST:=INT(NOT(BOOL(DEST) OR BOOL(SOURCE)))
  ELSE IF ICODE=10 THEN
            DEST:=INT(NOT(BOOL(DEST) AND BOOL(SOURCE)))
  ELSE IF ICODE=11 THEN
            DEST:=INT(NOT(BOOL(DEST) EQV BOOL(SOURCE)))
  ELSE IF ICODE=12 THEN
            DEST:=INT(BOOL(DEST) EQV BOOL(SOURCE))
```



```
  ELSE IF ICODE=13 THEN DEST:=SOURCE+DEST
  ELSE IF ICODE=14 THEN DEST:=SOURCE*DEST
  ELSE IF ICODE=15 THEN DEST:=DEST-SOURCE
  ELSE IF ICODE=16 THEN DEST:=SOURCE-DEST
  ELSE IF ICODE=17 THEN DEST:=DEST DIV SOURCE
  ELSE IF ICODE=18 THEN DEST:=SOURCE DIV DEST
  ELSE IF ICODE=19 THEN DEST:=DEST REM SOURCE
  ELSE IF ICODE=20 THEN DEST:=SOURCE REM DEST
  ELSE IF ICODE=21 THEN DEST:=DEST SOURCE
  ELSE IF ICODE=22 THEN DEST:=LN((X:=DEST))/LN((Y:=SOURCE))
  ELSE IF ICODE=23 THEN DEST:=CIR(SOURCE,DEST)
  ELSE IF ICODE=24 THEN DEST:=COMB(SOURCE,DEST)
  ELSE IF ICODE=25 THEN DEST:=IMAX(SOURCE,DEST)
  ELSE IF ICODE=26 THEN DEST:=IMIN(SOURCE,DEST)
  ELSE IF ICODE=27 THEN DEST:=SOURCE
  ELSE IF ICODE=28 THEN
  BEGIN
   ZCC:=SOURCE=DEST;
   GCC:=SOURCE DEST;
   LSB:=TF(BOOL(SOURCE) AND BOOL(DEST) AND %1);
  END;
  IF TRACE THEN
  BEGIN
    SPACE(4); PRINT(SOURCE,5); PRINT(DEST,5); NEWLINE;
  END;
END;


PROCEDURE SINGLE(LNUM,INSTR);  INTEGER LNUM;
      BOOLEAN INSTR;
      VALUE INSTR;
BEGIN  BOOLEAN  DF,DFMSK;   INTEGER ICODE,DFIELD;
  ICODE:= (INT(INSTR) DIV FIELDSIZE) REM SOPCODESIZE;
  DFIELD:= INT(INSTR) REM FIELDSIZE;
  IF DFIELD=0 THEN
  BEGIN
    NATPIDEST:=NATPIDEST+1;
    DNID:=NINSTRS-DNID;
    DNID:=IMIN(DNID,MAXHISTBINS);
    IBD[DNID]:=IBD[DNID]+1;
    DNID:=NINSTRS;
    DNI:=NINSTRS - DNI;
    DNI:=IMIN(DNI,MAXHISTBINS);
    IBMMREF[DNI]:=IBMMREF[DNI]+1;
    DNI:=NINSTRS;
  END;
  DF:=BOOL(DFIELD);  DFMSK:= NOT PARAMMASK;
  IF DF AND PARAMMASK THEN SXQT(ICODE,GDATA[DFIELD])
  ELSE IF DFIELD # 0 THEN SXQT(ICODE,LDATA[LNUM,DFIELD])
  ELSE SXQT(ICODE,DATA[LDATA[LNUM,1]]);
END;
```

```
PROCEDURE EXBRANCH(LNUM,INSTR);
   INTEGER LNUM;  BOOLEAN INSTR;    VALUE LNUM,INSTR;
BEGIN          INTEGER BCODE,DFIELD,SHIFTR;
  SHIFTR:=INT(BROFFSETMASK)+1;
  BCODE:=(INT(INSTR) DIV SHIFTR) REM 8;
  DFIELD:=INT(INSTR AND BROFFSETMASK);
  IF BCODE=0 THEN BRANCH(LNUM,DFIELD)
  ELSE IF BCODE=2 AND NOT ZCC THEN BRANCH(LNUM,DFIELD)
  ELSE IF BCODE=3 AND ZCC THEN BRANCH(LNUM,DFIELD)
  ELSE IF BCODE=4 AND NOT NCC THEN BRANCH(LNUM,DFIELD)
  ELSE IF BCODE=5 AND NCC THEN BRANCH(LNUM,DFIELD)
  ELSE IF BCODE=6 AND NOT(ZCC OR NCC) THEN
              BRANCH(LNUM,DFIELD)
  ELSE IF BCODE=7 AND ZCC OR NCC THEN
              BRANCH(LNUM,DFIELD);              !BNE;
  IF TRACE THEN
  BEGIN
    SPACE(4); WRITE("BRANCH"); PRINT(BCODE,4); NEWLINE;
  END;
END;
```

```
PROCEDURE DOUBLE(LNUM,INSTR);
      INTEGER LNUM;  BOOLEAN INSTR;
      VALUE LNUM,INSTR;
BEGIN   BOOLEAN SF,SFMSK,DF,DFMSK;
  INTEGER ICODE,SFIELD,SOURCE,DFIELD,T1,T2;
  ICODE:=INT(INSTR) DIV (FIELDSIZE*FIELDSIZE);
  SFIELD:= (INT(INSTR) DIV FIELDSIZE) REM FIELDSIZE;
  IF SFIELD=0 THEN
  BEGIN
    NATPISOURCE:=NATPISOURCE+1;
    DNIS:=NINSTRS-DNIS;
    DNIS:=IMINBDNIS,MAXHISTBINS);
    IBS[DNIS]:=IBS[DNIS]+1;
    DNIS:=NINSTRS;
    DNI:=NINSTRS - DNI;
    DNI:=IMIN(DNI,MAXHISTBINS);
    IBMMREF[DNI]:=IBMMREF[DNI]+1;
    DNI:=NINSTRS;
  END;
  DFIELD:= INT(INSTR) REM FIELDSIZE;
  IF DFIELD=0 THEN
  BEGIN
    NATPIDEST:=NATPIDEST+1;
    DNID:=NINSTRS-DNID;
    DNID:=IMIN(DNID,MAXHISTBINS);
    IBD[DNID]:=IBD[DNID]+1;
    DNID:=NINSTRS;
```



```
    DNI:=NINSTRS - DNI;
    DNI:=IMIN(DNI,MAXHISTBINS);
    IBMMREF[DNI]:=IBMMREF[DNI]+1;
    DNI:=NINSTRS;
  END;
  SF:=BOOL(SFIELD);  SFMSK:=NOT PARAMMASK;
  DF:=BOOL(DFIELD);  DFMSK:=SFMSK;
  SOURCE:= IF (SF AND PARAMMASK) THEN GDATA[SFIELD]
  ELSE IF SFIELD#0 THEN LDATA[LNUM,SFIELD]
  ELSE DATA[LDATA[LNUM,1]];
  IF DF AND PARAMMASK THEN
   DXQT(ICODE,SOURCE,GDATA[DFIELD])
   ELSE IF DFIELD # 0 THEN
     DXQT(ICODE,SOURCE,LDATA[LNUM,DFIELD])
     ELSE DXQT(ICODE,SOURCE,DATA[LDATA[LNUM,1]]);
END;
```

```
PROCEDURE EXECUTE(INSTR,LNUM);
      INTEGER LNUM; BOOLEAN INSTR;
      VALUE INSTR;
BEGIN
  NINSTRS:=NINSTRS+1;
  IF INSTR AND DOPMASK THEN DOUBLE(LNUM,INSTR) ELSE
  IF INSTR AND BRCODE THEN EXBRANCH(LNUM,INSTR) ELSE
  IF INSTR AND SOPMASK THEN SINGLE(LNUM,INSTR) ELSE
  IF INSTR AND ISTEPMASK THEN ISTEP(LNUM,INSTR)ELSE
  IF INSTR AND CRJMASK THEN CRJMP(LNUM,INSTR) ELSE
  IF MATCH(INSTR,HALT)  THEN RUN := FALSE ELSE
  BEGIN
    WRITE("ILLEGAL INSTRUCTION"); NEWLINE;
  END;
END;
```

```
PROCEDURE PRNTSUMMARY;
BEGIN   INTEGER K,I;
  NEWLINE(2);
  WRITE("THE NUMBER OF INSTRUCTIONS EXECUTED WAS");
  PRINT(NINSTRS,6);NEWLINE(2);
  WRITE("SOURCE REFERENCES TO MAIN MEMORY ");
  PRINT(NATPISOURCE,6);NEWLINE(2);
  IF NATPISOURCE#0 THEN
  BEGIN
    NEWLINE(2);
    WRITE("DISTRIBUTION OF INSTRUCTIONS BETWEEN THEM[N]");
    FOR I:=0 UNTIL MAXHISTBINS DO
    BEGIN
      K:=(100*IBS[I])/NATPISOURCE;   PRINT(K,4);
    END;
    NEWLINE(3);
```

```
    END;
    WRITE("DESTINATION REFERENCES TO MAIN MEMORY");
    PRINT(NATPIDEST,6);NEWLINE(2);
    IF NATPIDEST#0 THEN
    BEGIN
      NEWLINE(2);
      WRITE("DISTRIBUTION OF INSTRUCTIONS BETWEEN THEM[N]");
      FOR I:=0 UNTIL MAXHISTBINS DO
      BEGIN
        K:=(100*IBD[I])/NATPIDEST;     PRINT(K,4);
      END;
    END;
    IF NATPIDEST+NATPISOURCE#0 THEN
    BEGIN
      NEWLINE(3);
      WRITE("DIST OF LP INSTRS BETWEEN MEM REFS[N]");
      FOR I:=0 UNTIL MAXHISTBINS DO
      BEGIN
        K:=(100*IBMMREF[I])/(NATPIDEST+NATPISOURCE);
        PRINT(K,4);
      END;
    END;
  END;



INTEGER PROCEDURE LADR(FIX);
      INTEGER FIX;     VALUE FIX;
BEGIN
  LADR:=(FIX DIV 4096) REM 64;
END;



INTEGER PROCEDURE ITEM(FIX);
      INTEGER FIX;     VALUE FIX;
BEGIN
  ITEM:=(FIX DIV 64) REM 64;
END;



INTEGER PROCEDURE TEMP(FIX);
      INTEGER FIX;     VALUE FIX;
BEGIN
  TEMP:=FIX REM 64;
END;
```



```
  PROCEDURE INSTALLFIX;
  BEGIN
        INTEGER I;

    I:=0;
    WHILE FIXLIST[(I:=I+1)] # 0 DO
    LDATA[LADR(FIXLIST[I]),ITEM(FIXLIST[I])]:=
                GDATA[TEMP(FIXLIST[I])];
  END;



COMMENT  HERE IS THE MAIN PROGRAM;
BEGIN
  BOOLEAN INSTR;
  INTEGER CURLAD,PC,ICODE,ADR,I;
  INTEGER ARRAY ARG[1:3];
  SETUP;
  CURLAD:=STARTLAD;


  WHILE RUN DO
  BEGIN
    PC:=LDATA[CURLAD,2];
    INSTR:=CODE[PC];
    IF TRACE THEN
    BEGIN
      PRINT(CURLAD,5); PRINT(PC,10); SPACE(4);
      PRINTOCTAL(INSTR);
    END;
    LDATA[CURLAD,2]:=LDATA[CURLAD,2]+1;
    EXECUTE(INSTR,CURLAD);
  END;
  INSTALLFIX;
  OUTPUT(4,"DSK");  OPENFILE(4,"OUTPUT.DAT");
  SELECTOUTPUT(4);
  FOR I:=1 UNTIL NLADRS DO PRNTLAND(I);
  NEWLINE; PRINT(GDATA[35],4); NEWLINE;
END;
PRNTSUMMARY;
END;
```

REFERENCES

[Abrams 1970] P. S. Abrams. An APL Machine. Stanford University Computer Science Report STAN-CS-70-158, 1970.

[Abrams 1975] P. S. Abrams. What's Wrong With APL?, APL 75 Association for Computing Machinery, New York, 1975, 1-8.

[AMD 1] Schottky and Low-Power Schottky Bipolar Memory, Logic, and Interface. Advanced Micro Devices, Inc., Sunnyvale, Calif., (undated).

[AMD 1975] Am 2900 Bipolar Microprocessor Family. Advanced Micro Devices, Inc., Sunnyvale, Calif., June 1975.

[Bashkow 1967] T. R. Bashkow, A. Sasson, and A. Kronfeld. System Design of a Fortran Machine. IEEE Transactions on Computers, vol. EC-16, no. 4 (August 1967), 485-499.

[Bell 1971] C. G. Bell and A. Newell. Computer Structures: Readings and Examples. McGraw-Hill, New York, 1971.

[Bingham 1975] H. W. Bingham. Content Analysis of APL Defined Functions. APL 1975, Association for Computing Machinery, New York, 1975, 60-66.

[Blood 1972] W. R. Blood Jr. MECL System Design Handbook, 2nd ed., Motorola Semiconductor Products, Inc., Phoenix, Arizona, 1972.

[Breed 1967] L. M. Breed and R. H. Lathwell. The Implementation of APL/360. Proceedings of ACM Symposium on Interactive Systems for Experimental Applied Mathematics. ed. M. Klerer and J. Reinfelds. Academic Press, New York, 1968, 390-399.

[Burks 1954] A. W. Burks, D. W. Warren, and J. B. Wright. An Analysis of a Logical Machine Using Parenthesis-Free Notation. Mathematical Tables and Other Aids to Computation, vol. 8, no. 46, 53-57.

[Burroughs 1972] Burroughs B1700 System Reference Manual, Preliminary Edition. Burroughs Corp., Detroit, Mich., 1972.

[CD 1975] Protable Computer Combines Microprocessor Technology with Large-Computer Performance. Computer Design, vol. 14, no. 11 (November 1975), 130-131.

[DEC 1972] KB-11A Central Processor Unit Maintenance Manual. Digital Equipment Corp., Maynard, Mass., 1972.

[DEC 1975] PDP-11 04/05/10/35/40/45 Processor Handbook. Digital Equipment Corp., Maynard, Mass., 1975.

[DG 1972] How to Use the Nova Computers. Data General Corp., Southboro, Mass. 1972.

[Dusold 1975] L. Dusold. Letter to the Editor. APL Quote Quad, vol. 6, no. 3 (Fall 1975), 3.

[ED 1975] Full APL Computer Delivers Mainframe Power in Mini Size. Electronic Design, vol. 23, no. 24 (November 22, 1975), 171.

[Fairchild 1975] Full Line Condensed Catalog. Fairchild Semiconductor Components Group, Fairchild Camera and Instrument Corp., Mountain View, Calif., 1975.

[Gilman 1974] L. Gilman and A. J. Rose. APL An Interactive Approach, 2nd ed., John Wiley and Sons, New York, 1974.

[Goldstine 1972] H. H. Goldstine. The Computer from Pascal to von Neumann. Princeton University Press, Princeton, New Jersey, 1972.

[Grant 1974] C. A. Grant, M. L. Greenberg, and D. D. Redell. A Computer System Providing Microcoded APL. Proceedings of the Sixth International APL Users Conference, May 1974, 173-179.

[Hassitt 1972] A. Hassitt and L. E. Lyon. Efficeient Evaluation of Array Subscripts of Arrays. IBM Journal of Research and Development, January 1972, 45-57.

[Hassitt 1973] A. Hassitt, J. W. Layeschulte, and L. E. Lyon. Implementation of a High-Level Language Machine. Communications of the ACM, vol. 16, no. 4 (April 1973), 199-212.

[Hassitt 1975] A. Hassitt and L. E. Lyon. Direct Execution of APL on an IBM/370. APL 75. Association for Computing Machinery, New York, 183-191.

[Hodges 1976] D. A. Hodges. Trends in Computer Hardware Technology. Computer Design, vol. 15, no. 2 (February 1976), 77-85.

[Iverson 1962] K. E. Iverson. A Programming Language. John Wiley and Sons, New York, 1962.

[Keyes 1975] R. W. Keyes. Physical Limits in Digital Electronics. Proceedings of the IEEE, vol. 63, no. 5 (May 1975), 740-767.

[Mauchly 1947] J. W. Mauchly. Preparation of Problems for EDVAC-Type Machines, Proceedings of a Symposium on Large Scale Digital Calculating Machinery. In Annals of the Computation Laboratory of Harvard University, vol. 16, Harvard University Press, Cambridge, Mass., 1948, 203-207.

[Melbourne 1965] A. J. Melbourne and J. M. Pugmire. A Small Computer for the Direct Processing of FORTRAN Statements. The Computer Journal, vol. 8 (April 1965), 24-28.

[Motorola 1974] Semiconductor Data Library: Volume 4, MECL Integrated Circuits. Motorola Semiconductor Products, Inc., Phoenix, Arizona, 1974.

[Motorola 1976] Motorola Semiconductors Price List. Motorola Semiconductor Products, Inc., Phoenix, Arizona, Februrary 1976.

[Nanodata] Microprogramming the QM-1. Nanodata Corp. Williamsville, New York, (undated).

[Oakley 1975] J. D. Oakley. A comparison of Two Microprogrammable Processors: PDP-11/40E and MLP-900. Department of Computer Science, Carnegie-Mellon University Technical Report, May 1975.

[Perlis 1975] A. J. Perlis. Steps Toward an APL Compiler. Research Report #24, Department of Computer Science, Yale University, January 1974.

[Polivka 1975] R. P. Polivka and S. Pakin. APL: The Language and Its Usage, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[Randell 1975] B. Randell ed. The Origins of Digital Computers. Springer-Verlag, New York, 1975.

[Rosen 1967] S. Rosen. Programming Systems and Languages. McGraw-Hill, New York, 1967.

[Saal 1975] H. J. Sall and Z. Weis. Some Properties of APL Programs. APL 75. Association for Computing Machinery, New York, June 1975, 292-297.

[Sammett 1969] J. E. Sammett. Programming Languages: History and Fumdamentals, Prentice-Hall, Englewood Cliffs, New Jersey, 1969.

[Signetics 1970] Design of Microprogrammable Systems. Signetics Memory Systems, Sunnyvale, Calif. December 1970.

[Signetics 1971] Economic Advantages of Microprogramming. Signetics Memory Systems, Sunnyvale, Calif. January 1971.

[Tanenbaum 1976] A. S. Tanenbaum. Structured Computer Organization. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Thurber 1970] K. J. Thurber and J. W. Myrna. Systems Design of a Cellular APL Computer. IEEE Transactions on Computers, vol. C-19, no. 4 (April 1970), 290-303.

[TI 1973] The TTL Data Book for Design Engineers, Texas Instruments Inc., Dallas, Texas, 1973.

[TI 1974] Supplement to the TTL Data Book for Design Engineers. Texas Instruments Inc., Dallas, Texas, 1974.

[Weber 1967] H. Weber. A Microprogrammed Implementation of EULER of IBM System/360 Model 30. Communications of the ACM, vol. 10, no. 9 (September 1967), 549-558.

[Wilkes 1951a] M. V. Wilkes, D. J. Wheeler, and S. Gill. The Preparation of Programs for an Electronic Digital Computer. Addison-Wesley, Reading, Mass., 1951.

[Wilkes 1951b] M. V. Wilkes. The Best Way to Design an Automatic Calculating Machine. Manchester University Computing Inaugural Conference, July 1951, Published by Ferranti Ltd. London.

[Wilkes 1953] M. V. Wilkes and J. B. Stringer. Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer, Proc. Cambridge Phil. Soc., pt. 2, vol 49 (April 1953), 230-238.

[Zaks 1971] R. D. Zaks, D. Steingart, and J. Moore. A Firmware APL Time-Sharing System. AFIPS Conference Proceedings, vol. 38 (1971 SJCC), 179-190.