1987

Vector Pascal: a computer programming language for the FPS-164 array processor

Thomas Raymond Turner
Iowa State University

INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.
- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.
- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17"x 23" black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 6"x 9" black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.

Turner, Thomas Raymond

VECTOR PASCAL: A COMPUTER PROGRAMMING LANGUAGE FOR THE FPS-164 ARRAY PROCESSOR

Iowa State University PH.D. 1987

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __✓__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages ✓
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

_____

_____

University
Microfilms
International

Vector Pascal: A computer programming language for the FPS-164 array processor

by

Thomas Raymond Turner

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Electrical Engineering and Computer Engineering
Major: Electrical Engineering (Computer Engineering)

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1987

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Support for vector operations in computer programming languages is analyzed to determine if programs employing such operations run faster.

The programming language Vector Pascal is defined and compared to Fortran 8X and Actus. Vector Pascal contains definintions for matrix and vector operations and the Vector Pascal compiler translates vector expressions. The Vector Pascal compiler executes on an IBM Personal Computer AT and produces code for a Floating Point Systems FPS-164 Scientific Computer.

The standard benchmark LINPACK, which solves systems of linear equations, is transcribed from Fortran to Standard Pascal and Vector Pascal. The Vector Pascal version of LINPACK exploits vector operations defined in the language. The speedup of the Vector Pascal version of LINPACK over the Standard Pascal version is presented.

1. INTRODUCTION

The goal of this study is to quantify performance benefits that can be expected by applying vector computer programming language enhancements to algorithms based on linear algebra.

The procedure used to find performance benefits is: Select a prominent benchmark program that uses vector operations. Measure the execution time of the benchmark program. Alter the program to take advantage of vector language enhancements. Measure the execution time of the altered benchmark. Compute the speedup of the altered programs over the original. The speedup of the altered benchmark over the original is the performance benefit produced as the principal object of this study.

LINPACK [1] is a well known collection of FORTRAN subroutines used to solve systems of linear equations. LINPACK is also widely used as a benchmark for measuring how fast various computers can execute computation-intense programs. Because LINPACK is a prominent benchmark that uses vector operations, it was chosen as the primary vehicle to test the efficacy of matrix and vector language enhancements.

A language to write computer programs and a computer to execute the programs are necessary to conduct this study. A Floating Point Systems FPS-164 Scientific Computer was chosen for this investigation because it is specifically architected to perform vector operations. Designers of the FPS-164 chose pipelining as the principal technique for maximizing performance. Pipelined and other instructions are under the direct control of the programmer via micro-operations. The ability to issue micro-operations offers a challenge to the compiler writer to efficiently utilize such features which are usually hidden.

FORTRAN is the natural choice of a language to augment given the selection of LINPACK as the primary vehicle to test the effectiveness of vector language enhancements. In addition, the only compiler available for a FPS-164 was a FORTRAN compiler [2]. However, despite these compelling reasons, FORTRAN was not chosen. The size of the FORTRAN compiler (over 100,000 lines) and lack of design documentation make the compiler intractable material for modification.

Pascal programming language was selected because a small compiler (4,000 lines) with a good description of the design is available [3]. As a consequence of this choice, the LINPACK benchmark programs were transcribed from FORTRAN to Pascal.

The remaining chapters of the thesis are as follows:

Chapter 2 discusses previous work on computer languages to facilitate computation-intense computer programs, and shows how this study is related.

Chapter 3 describes enhancements to Pascal proposed in this thesis and discusses the similarity between proposed and previous work mentioned in Chapter 2. The version of Pascal produced in this thesis is called Vector Pascal.

Chapter 4 is devoted to the LINPACK benchmark. An overview of the theoretical foundation of LINPACK is presented. Two versions of LINPACK are discussed. The Standard Pascal transcription and the Vector Pascal version of LINPACK are described in detail.

Chapter 5 describes significant implementation aspects of the Vector Pascal compiler.

Chapter 6 contains timing and speedup data for the LINPACK benchmarks.

Chapter 7 contains conclusions which result from this study. The applicability of language extension to existing computation-intense programs is discussed.

2. PREVIOUS WORK

The object of this study is to determine if vector enhancements to a programming language can improve the performance of programs that use vectors. Architectural features in high performance computers which support vector operations, programming language design for such features, and compiler technology are areas of previous study which support this investigation. The Vector Pascal compiler produces instructions for a high performance computer, the FPS-164 Scientific Computer. The architectural features which support vector operations on the FPS-164 are discussed in the context of other high performance computers in section 2.1. Aspects of programming language design and extension are discussed in section 2.2. Performance metrics for Vector Pascal are evaluated in section 2.3 and an overview of the technology used to construct the Vector Pascal Compiler is given in section 2.4.

2.1 High Performance Computers

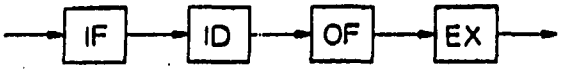
An overview of related computer structures is given here to place a FPS-164, for which the Pascal compiler described in this study generates instructions, in the family of computers. Pipelining is discussed because the FPS-164 employs software and hardware pipelines to achieve high performance and Vector Pascal expressions containing vector operands are translated into sequences of pipelined instructions. Hwang and Briggs [4] describe high-performance computers which include pipeline computers, array processors, and multiprocessor systems.

A nonpipelined computer executes instructions in four sequential steps: instruction fetch (IF) from main store; instruction decoding (ID), identifying and decoding operator and operand bits in the instruction; operand fetch (OF), retrieving the operand from the effective address calculated

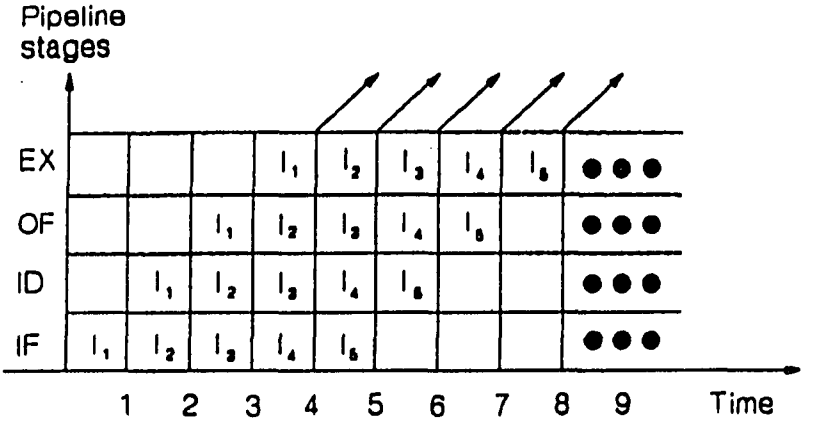
in the previous step; and execution (EX) where the decoded operation is performed. Each step must be completed before the next step is started.

A pipeline processor differs from a nonpipelined computer in that execution of successive instructions is overlapped. Figure 1, excerpted from Computer Architecture and Parallel Processing [4], shows instruction execution for pipelined and nonpipelined processors. A pipelined processor is illustrated in Figure 1 (a). Figure 1 (b) depicts the space-time diagram of a fully utilized pipelined processor. Five instructions are completed in eight time cycles. The space-time diagram for a nonpipelined processor is shown in Figure 1 (c). Two instructions are completed in eight time cycles.

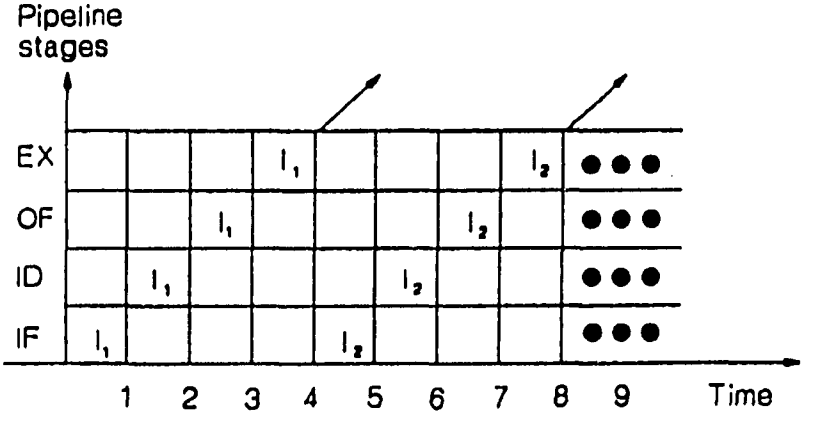
Efficient utilization of the capabilities of a pipeline processor is the key to extracting its maximum performance. Traditionally, one would say that a k-stage pipelined processor can be up to k times faster than an equivalent processor without pipelining. However, the author is not aware of a real processor that is exactly equivalent in all respects, save pipelining, to another with pipelining. Despite the problem of finding two comparable processors, the speedup due to pipelining can still be measured. In the FPS-164 pipelines are directly controlled by the programmer. Programs can be written so all operations complete before the next instruction is initiated and, conversely, programs can be written to exploit the pipelining features in the FPS-164 instruction set architecture. Utilization of the FPS-164 pipelines is characterized by the fraction of total execution time the pipelines are full. With the foregoing explanation the traditional statement limiting speedup due to pipelining can be changed to: A fully utilized k-stage pipeline processor can be up to k times faster than the



(a) A pipelined processor



(b) Space-time diagram for a pipelined processor



(c) Space-time diagram for a nonpipelined processor

Figure 1. Pipelined versus nonpipelined instruction execution

processor without taking advantage of its pipelining features.

CPU-bound instructions, like floating point operations, are also candidates for pipelining. Specific instruction distributions must exist, however, to warrant the inclusion of pipelined floating point hardware. A floating point instruction must occur repeatedly for such hardware to be effective. If it is known that an instruction will be executed repeatedly then the instruction fetch and decode steps can be eliminated: a vector processor uses this principle in vector instructions. The operand of a vector instruction identifies an array of data. One instruction is fetched to process many data elements in vector processors. To increase performance and decrease control logic complexity operands are often stored in vector registers.

Examples of vector processors include the Star 100 and Cyber 205 manufactured by Control Data, Cray-1 by Cray, and the VP-200 by Fujitsu. Attached pipelined processors include the FPS-164 by Floating Point Systems and the IBM 3838.

An array computer is a collection of arithmetic and logic units or processing elements (PE) which execute synchronously. A single control unit directs all processing elements and the same instruction is executed by all PEs. The Illiac-IV [5] and Burroughs Scientific Processor are examples of array computers.

Multiprocessor systems are characterized by Flynn [6] as Multiple-Instruction-stream-Multiple-Data-stream (MIMD) computers. Two or more processors share common main store, I/O channels, and peripheral devices. Local resources, including memory and private devices can be accessed only by a single processor. Communication between processors is accomplished via shared memories or by interrupts. The characteristic that distinguishes a multiprocessor system

from several computer systems participating in a local or long-haul network is the single, integrated operating system which manages all of the resources of the multiprocessor system. Examples of multiprocessor systems include CM* developed at Carnegie Mellon University [7,8], the IBM 3081, Denelcor HEP system, and the Cray X-MP and Cray-2 systems.

2.2 Pascal Language Extensions

The focus of this discussion is how to obtain the minimum execution time from a high level language program running on a high-performance computer. Motivation and purpose for extending Pascal and other languages are discussed first. One alternative to language extension based on dependency analysis is discussed. Vector and matrix features proposed in Actus and Fortran 8X are compared to similar features contained in Vector Pascal.

2.2.1 Language extension motivation and purpose

The precedent for extending existing languages and developing new languages is well established. Sammet [9] summarized many languages. The principal reason for suggesting new programming languages is that existing languages have insufficient power of expression. Jensen and Wirth [10] state their first reason for introducing Pascal "is to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language." Brinch-Hansen [11] created Concurrent Pascal to reduce the effort of programming operating systems. Backus [12], one of the originators of Fortran, argued for a functional programming language because "inherent defects at the most basic level cause [conventional programming languages] to be both fat and weak."

Another reason for language extension is that existing languages do not reflect the architecture of the underlying

computer. This is particularly true for vector processors. Investigators also hope that new languages will make it easier to exploit the parallel features of high-performance computers. New languages contain abstractions of the actual hardware and will therefore be easier to translate than strictly sequential program language statements. Reid and Wilson [13] write the following apology for the array features proposed in Fortran 8X.

> It is plain that array facilities are desperately needed in Fortran. Without them an efficiency conscious programmer may have to begin with vector or matrix operations, code them as DO loops, and then go to some trouble to ensure that the automatic vectorizer treats the innermost loops in vector mode. Such coding has been called "pornographic" by W. Kahan. Relief is on the way. The full language is unlikely to be in use until the 1990s, but vendors of vector and array machines are likely to implement subsets of the array features as extensions to Fortran 77.

A vision of researchers is: new vocabulary and expression will encourage programmers to develop parallel programs. Perrott [14] proposed Actus programming language to

> ... enable the specification of parallelism directly. The features are appropriate for both array and vector processors and they are defined without reference to the hardware of either type of machine; the algorithmic and data constructs are of sufficient generality and structure to make efficient use of the parallel computational resources.

Most conventional languages do not permit vector and matrix operations. Exceptions exist of course. Some implementations of BASIC have matrix and vector operations and APL has substantial support for operations that span many dimensions. However, these languages are, by and large, interpreted, and the performance of algorithms implemented in BASIC and APL is not a primary concern.

Other languages for vector processors include CFD, designed for the ILLIAC IV [15] and DAP-Fortran used on the

ICL Distributed Array Processor [16]: Perrott and Zarea-Aliabadi [17] explain differences between these languages. Both Fortran 8X and Actus are superior to CFD and DAP-Fortran. CFD, DAP-Fortran, and Actus support array declarations over which vector operations are defined. However, vector operations are valid for arrays containing exactly 64 elements in CFD and DAP-Fortran. The underlying computer architecture is reflected in the size of arrays which can be used in vector operations. Actus places no restrictions on array declarations based on the number of processing elements in a computer.

2.2.2 Dependence analyses of sequential programs

The foregoing discussion is based on the following methodology intended to improve the performance of high level language programs. Design programming languages that reflect and abstract the parallelism available in a computer's instruction set, and redesign computation-intense programs in the new language. An alternative is to design a compiler that will map sequential statements onto a computer capable of concurrent operations. The main benefit of the this approach is: existing programs need only be recompiled to run faster.

Compilers have been designed to recognize statements which can be executed concurrently and blocks of code which may be vectorized. Kuck et al. [18] developed techniques using dependence graphs to determine which statements in a program may be executed simultaneously. Kennedy [19] refined data dependence analyses which were successfully employed by Scarborough and Kolsky [20] to translate Fortran programs that exploit the vector instructions of an IBM 3090.

Most researchers interested in translating sequential languages into parallel or vector form for subsequent

execution have used Fortran in their investigations. Some reasons for choosing Fortran are explained below. Historically, Fortran has been used by the scientific community to implement computationally intense algorithms and, as a consequence, the scientific community has persistently requested more performance from programs written in Fortran. Supercomputer manufacturers usually make Fortran available and often Fortran is the only high level language for a particular supercomputer. Concurrent operations or vector instructions are almost always designed in supercomputers and available to the compiler writer to test translation algorithms. Finally, over years, a large investment has been made in Fortran applications. Applications would increase in value if techniques could be found, requiring no program changes, to execute them at the peak performance of supercomputers.

2.2.3 Comparison of Fortran 8X, Actus, and Vector Pascal

Compared to Fortran 8X and Actus, Vector Pascal has features which are limited to expressing algorithms based in linear algebra. Vector Pascal is not suited to signal processing applications. Vector Pascal does not permit trigonometric functions to accept vector operands or produce vector results. Vector Pascal does not support logical or circular shifts whereas both Fortran 8X and Actus do. Actus and Fortran 8X support a rich set of constructors which allow programmers to create banded, upper triangular, hermitian, and diagonal matrices: Vector Pascal has no such support.

Despite the limited scope of Vector Pascal it represents the essential character of languages designed for vector processors and can be used to measure performance benefits intrinsic to those languages.

2.3 Pascal Language Performance Metrics

The measurement of a programming language or extension is difficult. This study makes no attempt to devise a metric to measure the expressive power or consistent style of a programming language. Instead, this investigation attempts to shed insight on the performance improvements that can be realized by employing vector and matrix language features. This section explores metrics and suggests an appropriate measure for vector and matrix language constructs.

Siewiorek, Bell, and Newell [21] define how performance is measured. "Performance is measured in functions per unit of time or, conversely, the time needed to complete a specific function." Siewiorek, Bell, and Newell also suggest levels at which a computer can be measured. Performance levels include semiconductor physics, circuit, gate, register transfer, instruction set-processor (ISP), and system. At the ISP level the typical performance measure is the time to perform an instruction and the factor affecting performance is the sequence of register-transfer operations. At the system level the performance measure is is the time to perform an application and the factors at this level affecting performance are the instruction mix in the application, system software, system configuration, and the variation in input to the application.

Although a computer system is not being measured in this study, the performance measure for the system level is appropriate for measuring a programming language. A central point of this thesis is: the time to perform an application is a valid measure for a programming language. The execution time of an algorithm varies with the language in which it is implemented. This study suggests that the variation is not entirely due to compiler design.

Benchmark programs have been used extensively to measure the performance of computer systems [22,23,24]. Siewiorek, Bell, and Newell suggest standard benchmarks "reflect the application as well as characterizing the language machine architecture." A relevant application that can be used to test Vector Pascal is LINPACK [1]. LINPACK is appropriate because it uses standard algorithms of linear algebra and it represents typical applications executed on high-performance computers. LINPACK is also widely used as a benchmark to measure the number of floating point operations per second that a computer can execute.

2.4 Pascal Language Implementation Technology

Many have contributed to the field of compiler construction including those used in this study [3,25,26,27,28,29]. Pemberton and Daniels [3] describe the P4 compiler used in this investigation. Urs Ammann, Kesav Nori, and Christian Jacobi designed the compiler. The compiler employs syntax-directed methods and a top-down recursive descent parser is used to analyze Pascal programs.

The P4 compiler translates standard Pascal [10] to P-code, a hypothetical stack-oriented computer for Pascal. The P4 compiler was extended to recognize the extended syntax described in Chapter 3. Instructions to support vector operations have been added to the original P-code definition. P-code instructions are translated to APAL64 [30], the instruction set of a FPS-164 by an assembler developed in this study. The extended P4 compiler and assembler execute on an IBM Personal Computer AT under the control of DOS 3.1.

3. VECTOR PASCAL

Additions to the Pascal programming language proposed in this section are to assist programmers express algorithms based on the mathematics of linear algebra [31] conveniently and to permit the compiler writer to generate efficient programs using syntax-directed methods.

Motivation for extensions to Standard Pascal originates in archetypical applications which employ techniques founded in linear algebra. An application of linear algebra is solving systems of linear equations. LINPACK [3] is a well-known package of subroutines which finds the solution vector x in the matrix-vector equation

$$Ax = b.$$

Gaussian elimination is used to factor matrix A into upper and lower triangular form. Elements, in rows, to the right of the diagonal are manipulated as the matrix is factored. Column-elements, below the diagonal, are searched to find the element having the largest magnitude. These operations suggest the need to operate on slices of a matrix, in other words, portions of rows or columns of the matrix.

Extensions added to Vector Pascal are similar to vector and matrix manipulations available in the Ada programming language. A discussion of vector and matrix operations implemented in Ada can be found in Chapter 3 of Ada for Experienced Programmers [32]. Table 1 contains declarations for vector and matrix operations defined in Tables 2, 3, 4, and 5.

Declarations in Table 1 are discussed in the following paragraphs.

Constant N is bound to the integer value 100 at translation-time. Constant N is the number of elements in a vector and the rank of a matrix.

Table 1. Vector Pascal declarations

Declaration	Description
const	
N = 100;	Size of rows and columns
type	
idxrng = 1..N;	Range over which indexes may vary
ivector = array[idxrng] of integer;	Integer vector type
imatrix = array[idxrng] of ivector;	Integer square matrix type
rvector = array[idxrng] of real;	Real vector type
rmatrix = array[idxrng] of rvector;	Real square matrix type
var	
r,c,o : idxrng;	Row, column subscripts
k : integer;	Integer scale factor
l,m,p : ivector;	Integer vectors
H,I,J : imatrix;	Integer matrices
s : real;	Real scale factor
a,b,x : rvector;	Real vectors
D,E,F : rmatrix;	Real matrices

Type idxrng identifies the valid range over which indexes to vectors and matrices can vary.

Type ivector describes an integer array. Both Standard Pascal operations on individual elements and Vector Pascal operations on the entire array are valid in Vector Pascal. Type idxrng identifies the range of valid indexes. There are 100 (N) elements in a vector of type ivector. The first valid index is one and the last is 100.

Type imatrix is a template for an array of integer vectors. Each integer vector is described by the type ivector. The range and number of integer vectors is controlled by type idxrng. There are 100 integer vectors each having 100 elements. Type imatrix describes a square, 100 by 100, matrix containing integer elements.

Type rvector is analogous to type ivector. Type rvector describes an array of real elements. There are 100 elements numbered 1 through 100. Type idxrng describes the range and type of variables which can be used to index an array of type rvector.

Type rmatrix describes a real matrix having 100 rows and 100 columns. Type rmatrix is the real analogue of type imatrix. The type declaration literally specifies an array of real vectors. Each real vector is specified by type rvector. The range over which indexes may vary is controlled by type idxrng.

Variables r, c, and o, are used in subsequent illustrations as indexes to integer and real vectors and matrices.

Variable k is used to scale integer vectors and matrices in Tables 2 and 4.

Vectors l, m, and p are used to illustrate valid integer vector operations in Tables 2 and 4.

Matrices H, I, and J are used in Table 4 to depict matrix operations available in Vector Pascal.

Variable s is used to scale real vectors and matrices in Tables 3 and 5.

Real vectors a, b, and x are used in Tables 3 and 5 to show operations with real vectors and matrices available in Vector Pascal.

Variables D, E, and F are used in the description of operations valid for real matrices depicted in Table 5.

Table 2 illustrates all operations valid for integer vectors in Vector Pascal. The left column lists the Vector Pascal syntax for the operation defined by Standard Pascal statements in the middle column. A brief description appears in the right column. Four operations, scalar multiplication, inner product, vector addition, and vector subtraction, are illustrated in Table 2.

Table 2. Integer vector operations

Vector Pascal	Standard Pascal	Description
`l := k*m;`	`for r:=1 to N do` `  l[r] := k*m[r];`	scalar multiplication

`k := m*p;`	`k:=0;`
	`for r:=1 to N do`
	`  k := k + m[r]*p[r];`
		inner product

`l := m+p;`	`for r:=1 to N do` `  l[r] := m[r]+p[r];`	vector addition
`l := m-p;`	`for r:=1 to N do` `  l[r] := m[r]-p[r];`	vector subtraction

Every element in a vector is multiplied by a variable of the same type in the scalar multiplication operation. Variable k is the scalar used to multiply every element of integer vector m. The expression "m scaled by k" is often used to describe the foregoing process. Vector l is assigned the result of the vector expression.

The inner product is the sum of the products of corresponding elements. Corresponding elements have the same index. The result of an inner product is a scalar. Variable k is assigned the result of the inner product of integer vectors m and p. The inner product is sometimes called the "dot product."

Vector addition produces a vector containing the sum of corresponding elements in the addend and augend.

Vector subtraction produces a vector containing the difference of corresponding elements in the minuend and subtrahend.

Real vector operations described in Table 3 are analogous to the integer vector operations described above.

Table 4 illustrates all operations valid for integer matrices in Vector Pascal. Integer matrix operations as defined in Table 4 are not implemented in this study. The left column lists the Vector Pascal syntax for the operation defined by Standard Pascal statements in the middle column.

A brief description appears in the right column. Five operations, scalar multiplication, vector-matrix multiplication, matrix-vector multiplication, matrix addition, and matrix subtraction, are illustrated in Table 4.

Table 3. Real vector operations

Vector Pascal	Standard Pascal	Description
a := s*b;	for r:=1 to N do a[r] := s*b[r];	scalar multiplication

s := b*x;	s:=0;
	for r:=1 to N do
	  s := s + b[r]*x[r];
		inner product

a := b+x;	for r:=1 to N do a[r] := b[r]+x[r];	vector addition
a := b-x;	for r:=1 to N do a[r] := b[r]-x[r];	vector subtraction

Scalar multiplication for matrices is similar to scalar multiplication for vectors. Every element of the matrix I is multiplied by the integer scale factor k. The resulting matrix is assigned to matrix H.

The second and third operations have a matrix and a vector as operands. The result in both cases is a vector.

In vector-matrix multiplication, vector m can be thought of as a row-vector. Elements of vector l consist of the inner product of the row-vector m with the corresponding columns of matrix H. Corresponding means that the element of l receiving the inner product and the column of H used as an operand in the inner product have the same index.

In matrix-vector multiplication, the vector should be considered an column-vector. Elements of the result vector, l, are the inner products of corresponding rows of matrix H and the column-vector m.

Table 4. Integer matrix operations

Vector Pascal	Standard Pascal	Description

`H := k*I;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  H[r,c] := k*I[r,c];`
		scalar multiplication

`l := m*H;`	`for c:=1 to N do`
	`  begin l[c] := 0;`
	`  for r:=1 to N do`
	`  l[c] := l[c] +`
	`  m[r]*H[r,c];`
	`  end`
		vector-matrix multiplication

`l := H*m;`	`for r:=1 to N do`
	`  begin l[r] := 0;`
	`  for c:=1 to N do`
	`  l[r] := l[r] +`
	`  H[r,c]*m[c]`
	`  end;`
		matrix-vector multiplication

`H := I+J;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  H[r,c] := I[r,c] +`
	`  J[r,c];`
		matrix addition

`H := I-J;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  H[r,c] := I[r,c] -`
	`  J[r,c];`
		matrix subtraction

`H := I*J;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  begin k := 0;`
	`  for o := 1 to N do`
	`  k := k + I[r,o]*J[o,c];`
	`  H[r,c] := k`
	`  end;`
		matrix multiplication

Matrix addition produces a matrix containing the sum of corresponding elements in the addend and augend.

Matrix subtraction produces a matrix containing the difference of corresponding elements in the minuend and subtrahend.

Real matrix operations shown in Table 5 are similar to integer matrix operations described above. Real matrix operations as defined in Table 5 are not implemented in this study.

Table 5. Real matrix operations

Vector Pascal	Standard Pascal	Description

`D := s*E;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  D[r,c] := s*E[r,c];`
		scalar multiplication

`b := x*D;`	`for c:=1 to N do`
	`  begin b[c] := 0;`
	`  for r:=1 to N do`
	`  b[c] := b[c] +`
	`  x[r]*D[r,c];`
	`  end`
		vector-matrix multiplication

`b := D*x;`	`for r:=1 to N do`
	`  begin b[r] := 0;`
	`  for c:=1 to N do`
	`  b[r] := b[r] +`
	`  D[r,c]*x[c]`
	`  end;`
		matrix-vector multiplication

`D := E+F;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  D[r,c] := E[r,c] +`
	`  F[r,c];`
		matrix addition

`D := E-F;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  D[r,c] := E[r,c] -`
	`  F[r,c];`
		matrix subtraction

`D := E*F;`	`for r:=1 to N do`
	`  for c:=1 to N do`
	`  begin s := 0;`
	`  for o := 1 to N do`
	`  s := s + E[r,o]*F[o,c];`
	`  D[r,c] := s`
	`  end;`
		matrix multiplication

3.1 Slices

A slice is a contiguous portion of a vector. A slice of a vector or matrix is identified by the range syntax, <range> ::= <expression> .. <expression>. Table 6 illustrates the use of slices.

The example in Table 6 is taken from the transcription of LINPACK produced in this study. Subroutine DGEFA uses Gaussian elimination to factor matrix A. The inner loop produces zeros below the diagonal in column c. The variable

pivotrow retains the index of the pivot row. Function IMAX finds the index of element having the largest magnitude. The argument to IMAX is a column-slice of matrix A in column c beginning with the row on the diagonal and terminating at the end of the column. IMAX returns a value in the range c to rank. A[r,c] is the factor by which the pivot row is scaled to produce a zero in column c, row r. Row-slices of matrix A are added to the pivot row, properly scaled by A[r,c], in the inner loop. The slice begins one element to the right of the diagonal and terminates at the end of the row.

A matrix cannot be sliced in two dimensions. The range syntax can appear as a subscript in only one index position.

Table 6. Slices

Declaration	Description
`type`	
`  idxrng  = 1..rank;`	Range over which indexes may vary
`  rvector =`	Real vector type
`    array[idxrng]`	
`    of real;`	
`  rmatrix =`	Real matrix type
`    array[idxrng]`	
`    of rvector;`	
`var`	
`  A       : rmatrix;`	A is a real matrix
`  r,c     : idxrng;`	r and c are row and column indexes
`  pivotrow: idxrng;`	Used to retain the pivot row index
`. . .`	
`  for c:=1 to rank-1 do`	Begin Gaussian elimination
`    begin`	
`      pivotrow :=`	Extract the index of the element
`      IMAX(A[c..rank,c]);`	in column c having the largest magnitude
`      . . .`	
`      for r:=pivotrow+1`	Put zeros in column c below the
`        to rank do`	pivot row
`        A[r,c+1..rank] :=`	
`        A[r,c+1..rank] +`	
`        A[r,c]*A[pivotrow,c+1..rank];`	
`    end;`	

Table 7. Invalid use of slices

Declaration	Description
`type`	
`  idxrng  = 1..N;`	Range over which indexes may vary
`  rvector =`	Real vector type
`    array[idxrng]`	
`    of real;`	
`  rmatrix =`	Real matrix type
`    array[idxrng]`	
`    of rvector;`	
`var`	
`  A       : rmatrix;`	A is a real matrix
`  x,b     : rvector;`	x and b are real vectors
`begin`	
`  x[5..10] := A[5..10,5..10]*b[5..10];`	

The expression A[5..10,5..10] is incorrect because it identifies a two-dimensional 6X6 partition of A rather than a one-dimensional vector. Slices are one-dimensional aggregates.

3.2 Open Array Parameters

Array bounds are assigned to array variables at translation-time in Standard Pascal. This restriction prevents the implementation of generic procedures that may perform operations defined for arrays of any size. For example, one procedure which computes the sum of two conforming matrices of various sizes cannot be implemented. Procedures for every different index range must be coded to implement matrix addition.

The purpose of open array parameters is to permit generic operations, implemented as procedures or functions, where the array bounds are passed implicitly with the array. Another discussion of open array parameters can be found in Programming in Modula-2 [33].

Arrays of any size may be substituted as arguments to procedures having open array parameters. Open array

parameters can only appear as formal parameters. Table 8 illustrates their use.

Table 8. Open array parameter usage

Declaration	Description

```
type
  idxrng  = 1..N;          Range over which indexes may vary
  rvector =                Real vector type
    array[idxrng]
    of real;
  rvectorparm =            Real vector parameter
    array of real;
  var
    A,B,C: rvector;        A, B, and C are real vectors
  procedure VecAdd         Procedure VecAdd is equivalent to
                            the Vector Pascal statement
                            Sum := Augend + Addend;
    (var Sum,Augend
    ,Addend: rvectorparm);
    var
      r: idxrng;
   begin
     for r := Low(Augend) to High(Augend) do
       Sum[r] := Augend[r] + Addend[r];
    end{vecadd};
begin
  . . .
  vecadd(A,B,C);
end.
```

The lower bound of a formal open array parameter is obtained by using the standard function Low. The upper bound can be found by employing Vector Pascal's standard function High. In Table 8, Sum is an array and Low(Sum) returns the value 1. Similarly, High(Sum) returns the value N.

3.3 Vector Pascal Syntax Diagrams

Vector Pascal syntax is characterized by Standard Pascal syntax diagrams found on pages 116-118 of the Pascal User Manual and Report [10] with the following exceptions. The type and variable syntax diagrams are modified as shown

in Figures 2 and 3. The syntax diagram for index expression in Figure 4 shows the correct use of slices. The altered type syntax diagram permits the declaration of open array parameters.

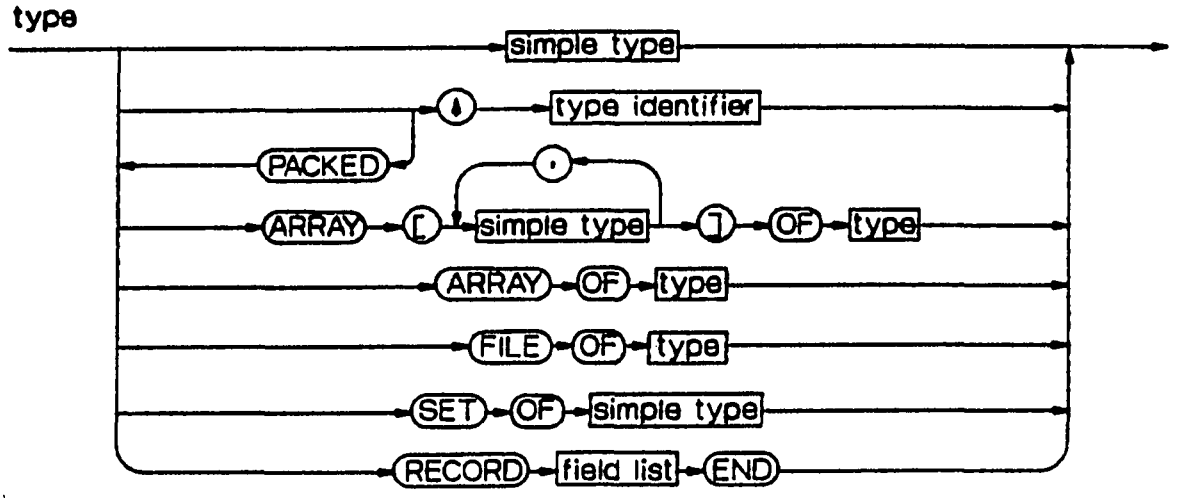


Figure 2. Type syntax diagram

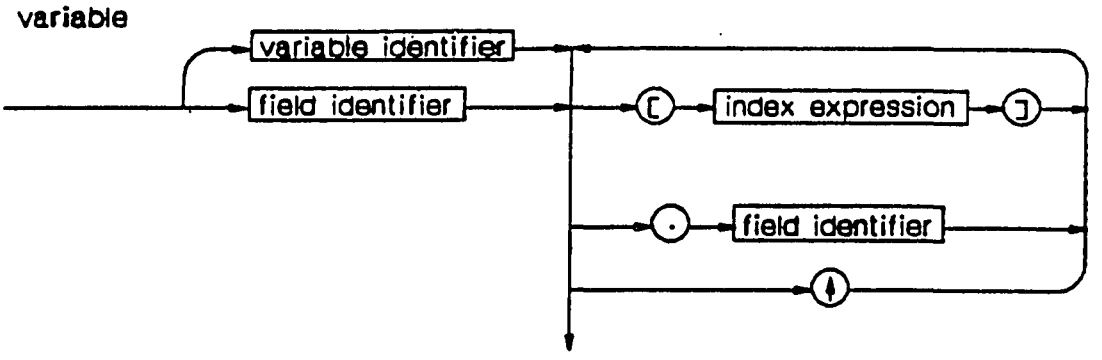


Figure 3. Variable syntax diagram

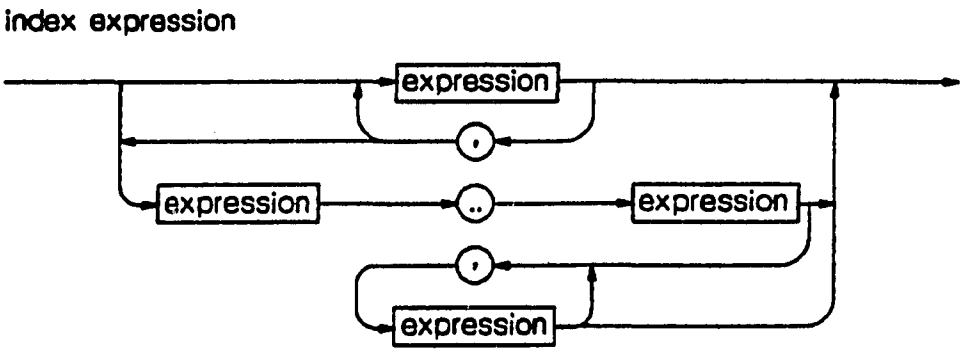


Figure 4. Index expression syntax diagram

3.4 Similarities of Vector Pascal, Actus, and Fortran 8X

Examples of vector declarations and operations as they are defined in the programming language Actus and the proposal for Fortran 8X are illustrated in this section. Comparisons between Vector Pascal, Fortran 8X, and Actus are given. The purpose of this section is to convince the reader that there is a strong resemblance between the three languages, and an analysis of performance benefits derived from using these languages is needed.

Table 9 lists a few examples of parallel declarations permitted in Actus. These declarations are used in subsequent Tables in which expressions and statements permissible in Actus are illustrated. Parallel constants subrange, interval, and amorphous are declared at the top of the Table. Sets, defined as an INDEX, can be used to index vectors. Both parallel constants and parallel indexes are multi-valued. Variables are declared after the VAR reserve word as they are in Standard Pascal. The way array bounds are declared differs from Standard Pascal. A colon between the low bound and the high bound indicates all elements in that dimension may be referenced in parallel.

Table 9. Actus parallel declarations

Declaration	Description

PARCONST
subrange = 10:30;
interval = 3:[5] 18;
amorphous = 5:[-2] 1,
13:17;
	Reserve word PARCONST marks the beginning of parallel constants. Examples of three variations are given. Constant subrange illustrates the sequence [10,11,12,...,30]. Interval represents a sequence with a constant interval between elements [3,8,13,18]. Amorphous concatenates two unrelated sequences [5,3,1,13,14,15,16,17].

INDEX ndxrng = 1:rank; ndxmix = 5:[-2] 1, 13:15;	Reserve word INDEX denotes the start of predefined sets of indices which may be used to subscript arrays. For example, if V is an array then V[ndxrng] = V[1],V[2],...,V[rank]. Rank must a defined constant. V[ndxmix] = V[5],V[3],V[1],V[13],V[14],V[15]
VAR vector,v1,v2: array [1:rank] of integer; matrix: array[1..rank, 1:rank] of integer;	Reserve word VAR identifies variable declarations. The colon (:) between the low bound, 1, and the high bound, rank, indicates all elements in that dimension will be referenced in parallel. The range token (..) directs the compiler to dereference all elements in that dimension sequentially.

A few expressions, highlighting vector operations available in Actus, are shown in Table 10. Refer to Table 9 for a description of the variables used in these expressions. The first two expression illustrate how the plus sign yields a vector result when one operand is a vector. The latter two expression show the use of standard function SUM. The last expression in Table 10 illustrates one way to compute an inner product using the facilities of Actus.

Table 10. Actus parallel expressions

Expression	Description
vector[ndxrng] + 5	The integer value 5 is added to every element of vector.
vector[10:15] + 5	The integer value 5 is added to elements of array vector in the range vector[10] through vector[15].
SUM(vector[3:[3] 9]	Standard function SUM returns the scalar sum of every third element starting with vector[3] and ending with vector[9]. SUM is generic:
If	vector had elements of type real then SUM would return type real. In this example vector has elements of type integer, so SUM is an integer function.
SUM(v1*v2)	Inner product of v1 and v2. The product of corresponding elements of v1 and v2 are computed. The scalar sum of products is returned by SUM.

Selected statements which focus on the vector features of Actus are listed in Table 11. Refer to Table 9 for variables used in these statements. The first statement illustrates vector assignment. The second and third statements show the use of the SHIFT and ROTATE operators. The last statement shows how the # operator takes on an index set determined by a vector Boolean expression.

The expression which appears as subscript in a vector is a set and set operations may expand or reduce the extent of the parallel operation in Actus.

Whereas Actus depends on variable declarations to identify the scope of parallelism Fortran 8X does not. Fortran 8X makes extensive use of intrinsic functions to define the boundaries of matrix and vector operations. Examples of such functions are in the following Tables.

Table 11. Actus parallel statements

Statement	Description

vector[10:30] := subrange;	Sequential elements of the array vector are assigned the value of their corresponding subscripts.
	vector[10] := 10;
	vector[11] := 11;
	...
	vector[30] := 30;

vector[5:10] := matrix[k,15:20 SHIFT 5];	The SHIFT operator adds the index value to the right of the operator (5) to every subscript in the source range.
	vector[5] := matrix[k,20];
	vector[6] := matrix[k,21];
	...
	vector[10] := matrix[k,25];

vector[5:8] := vector[1:4 ROTATE -2];	The ROTATE operator forms a ring bounded by the index pair to the left of the operator. Elements are chosen by adding the rotation constant to the current index value within the ring.
	vector[5] := vector[3];
	vector[6] := vector[4];
	vector[7] := vector[1];
	vector[8] := vector[2];

IF vector[1:4] > 0
THEN vector[#] := 0
ELSE vector[#] := 1;
	Every element in the range 1 through 5 greater than zero is set to one. All other elements in the range are set to zero. Assuming elements 1 through 4 of vector are assigned -1,0,1, and 2 respectively before the IF statement, then after the IF statement,
	vector[1] := 1;
	vector[2] := 1;
	vector[3] := 0;
	vector[4] := 0;
	The # operator takes on the index set of the elements causing the boolean expression to evaluate true in the THEN clause and the index set of the elements causing the boolean expression to be false in the ELSE clause.

Table 12. Fortran 8X declarations

Declaration	Description
REAL V1(N),V2(N),V3(N)	V1, V2, and V3 are N-tuples composed of real numbers.
REAL M1(N,M),M2(M,P)	M1 is a real matrix of N rows and M columns. M2 is a real matrix of M rows and P columns.

Declarations which conform to the Fortran 8X proposal are shown in Table 12. These declarations are used in examples of Fortran 8X intrinsic functions in Table 13 and statements in Table 14. Note that these declarations are also valid in Fortran 77.

Table 13. Fortran 8X intrinsic functions

Intrinsic Function	Description
DOTPRODUCT(V1,V2)	Returns a real scalar, the inner product of V1 and V2
MATMUL(M1,M2)	Returns the product of matrices M1 and M2, a real matrix having N rows and P columns.
MAXVAL(M1)	Returns the maximum value in the matrix M1.
SUM(V1*V2,MASK=V1.GT.1.)	Returns the sum of the product of corresponding elements in V1 and V2. The product of corresponding elements is computed only if the element in V1 is greater than 1. Assume V1 = (-1,0,1,2) and V2 = (1,1,1,1) then SUM will return (0) + (0*1) + (1*1) + (2*1) = 3.
HBOUND(V1)	Returns the maximum index of V1, N.
LBOUND(V1)	Returns the minimum index of V1, 1.

Examples of selected functions proposed in Fortran 8X are shown in Table 13. Descriptions of these functions appear beside the examples. Among the functions shown are

those which perform the inner product and matrix multiplication.

Examples of Fortran 8X vector assignment statements are shown in Table 14. Control over individual elements is shown in the second example. Matrix constructors are illustrated in the third example.

Table 14. Fortran 8X statements

Statement	Description
V3 = V1 + SIN(V2)	Elements of V1 are added to the sine of corresponding elements of V2 and sums are assigned to elements of V3.
WHERE (V1.GT.0) V2=V1	Positive elements of V1 are assigned to corresponding elements of V2.
M1(2,5:9)=[3[1],2[2]]	Elements 5, 6, and 7 of row 2 in matrix M1 are assigned the value 1. Elements 8 and 9 are set to the value 2.

The proposed Fortran 8X contains many more functions than those described here. Capabilities to test values in logical matrices are available. The product of array elements can be computed. Explicit dynamic memory management tailored for temporary vectors and matrices is proposed. Partitions of larger aggregates can be obtained. The number of rows and columns can be dynamically adjusted and matrices can be merged together. Reid and Wilson [13] present a summary of the array features in Fortran 8X.

The array features of Vector Pascal are similar to more extensive features in Actus and Fortran 8X. The essential character of all three languages is the same. All seek to provide syntax that facilitates the expression of computation-intense algorithms and that can be translated into efficient code for pipelined and vector computers.

Vector Pascal, Actus, and Fortran 8X all have the ability to express well defined operations on vectors without employing for-loops to explicitly enumerate component scalar instructions.

To perform vector operations both vectors and corresponding operations need to be identified. Vectors are referenced as simple arrays in Vector Pascal and Fortran 8X. Vector operations are defined only for real and integer vectors in Vector Pascal. Table 1 shows Standard Pascal type and variable declarations which are used in subsequent Tables to illustrate vector and matrix operations permitted in Vector Pascal. Logical vector operations are possible in Fortran 8X as well as standard real and integer vector operations. Table 12 lists Fortran variable declarations used in vector and matrix operations illustrated in Tables 13 and 14.

The range syntax in Vector Pascal allows designers to choose a contiguous portion of the vector if the entire vector cannot be used. Similarly, the colon is used in Fortran 8X and Actus to identify sequential elements of a vector.

Vectors are differentiated from arrays in Actus. Vectors are declared by inserting a colon between the low-index bound and the high-index bound. Arrays are known by the range (..) token separating the low-index bound from the high-index bound. Vector operations are defined on vectors but not on arrays in Actus. Only scalar operations are permitted on array elements. Integral operations involving all the elements of an array are valid for vectors but erroneous for arrays. For example, the inner product operation, implemented via the SUM standard function shown in Table 10 is defined for conforming vectors but is not defined on arrays in Actus.

3.5 Summary

This chapter described enhancements to Standard Pascal which define Vector Pascal and compared Vector Pascal to Actus and Fortran 8X. Fortran 8X and Actus have many more features than Vector Pascal but all three languages share central themes. All three languages are intended for use on a pipelined or vector computer. All three languages have abstracted the characterization of a vector found in the computer's instruction set to that appropriate for a high level language. Vector operations expressed in all three languages are designed to produce efficient code.

It is the purpose of this study to give an estimate of the speedup over traditional approaches that can be obtained by employing vector operations as defined in Fortran 8X, Actus, Vector Pascal, and similar languages.

4. LINPACK

The purpose of this chapter is to review the mechanics of solving systems of linear equations by lower-upper (LU) factorization and to present two transcriptions of LINPACK subroutines which factor and solve matrices.

4.1 LU Factorization

A canonical representation of a system of linear equations is shown in Table 15. Using matrix notation a system of linear equations is expressed $Ax = b$ where the coefficient matrix A is composed of a_{ij}, $1 <= i,j <= n$, and the right hand side vector b has elements b_i, $1 <= i <= n$. The problem is to find x_i, $1 <= i <= n$, the solution vector x. Table 16 illustrates the matrix notation.

Table 15. A system of linear equations

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots & \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
$$

Table 16. Matrix notation

$$
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & & \vdots \\
a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix}
\begin{bmatrix}
x_1 \\ x_2 \\ \vdots \\ x_n
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\ b_2 \\ \vdots \\ b_n
\end{bmatrix}
$$

Matrix A is square having n rows and columns. Column vectors x and b have n elements. The system of linear equations shown in Table 15 is produced by equating the inner product of the rows of A and vector x to the elements of b. An inner product is the sum of the products of row-elements and corresponding elements of x. Corresponding elements are chosen by matching the column subscripts of the row with the (column) subscripts of vector x. The first equation in Table 14 is produced by equating the inner product of the first row of A and vector x to the first element of b. In a similar way the second equation is obtained by setting the inner product of the second row of A and vector x equal to b_2. Subsequent equations are produced by equating the inner products of the remainder of the rows of A and vector x to b_3, b_4, ..., b_n respectively.

The solution to the system of linear equations would be quite easy to obtain if the coefficient matrix were in upper triangular form as shown in Table 17.

Table 17. Upper triangular form

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \cdot \\ \cdot \\ \cdot \\ y_n \end{bmatrix}$$

Substituting U for A and y for b the system of linear equations becomes $Ux = y$. The nth equation of the linear system in Table 17 is

$$u_{nn}x_n = y_n$$

and the solution to that equation is

$$x_n = y_n/u_{nn}.$$

The equation derived from the n-1 row of U is

$$u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = b_{n-1}$$

and the solution, using x_n found by solving the nth equation, is

$$x_{n-1} = (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1}$$

Each new equation, moving from the bottom row toward the top, introduces exactly one new unknown element of x which can be found using the elements of x discovered in previous equations. This method of finding the solution to an upper triangular matrix is called backward substitution because the elements of the solution vector are found in reverse order starting with x_n. Table 18 lists the general solution to a matrix in upper triangular form using backward substitution.

Table 18. Equations for backward substitution

$x_n = y_n/u_{nn}$
$x_i = (b_i - (u_{i,i+1}x_{i+1} + u_{i,i+2}x_{i+2} + \ldots + u_{i,n}x_n))/u_{ii}$ $1 <= i <= n-1$

Backward substitution is used to find the solution vector x for the upper triangular matrix of Table 19. The solution to x_3 is computed first, then x_2 and x_1.

Suppose the solution, y, to the matrix equation

$$Ly = b$$

Table 19. Backward substitution example

$$\begin{bmatrix} 4 & 1 & 3 \\ 0 & 7.5 & -0.5 \\ 0 & 0 & 4.5 \end{bmatrix} \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 7 \\ 4.5 \end{bmatrix}$$

$x_3 = y_3/u_{33} = 4.5/4.5 = 1$

$x_2 = (y_2 - (u_{23}x_3))/u_{22} = (7 - (-0.5*1))/7.5 = 1$

$x_1 = (y_1 - (u_{12}x_2 + u_{13}x_3))/u_{11} = (8 - (1*1 + 3*1))/4 = 1$

was desired where L was a unit lower triangular matrix as shown in Table 20. A matrix in unit lower triangular form contains zeros above the diagonal, ones along the diagonal, and arbitrary values, l_{ij}, $2 <= i <= n$, $1 <= j < i$, below the diagonal. A process similar to backward substitution, called forward substitution, can be used to find y.

Table 20. Unit lower triangular matrix

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ \cdot & \cdot & \cdot & \cdots & \cdot \\ l_{n1} & l_{n2} & l_{n3} & \cdots & 1 \end{bmatrix}$$

Forward substitution is denoted forward because elements of y are computed starting with y_1.

$$y_1 = b_1$$

y_2 is found using y_1.

$$y_2 = b_2 - l_{21}y_1$$

Table 21 lists equations for solving matrices in unit lower triangular form.

Table 21. Equations for forward substitution

$$y_1 = b_1$$

$$y_i = b_i - (l_{i1}y_1 + l_{i2}y_2 + \ldots + l_{ik}y_k)$$

$$1 < i <= n,\ k = i - 1$$

Table 22 illustrates the process by which a matrix in unit lower triangular form is solved.

Table 22. Forward substitution example

$$\begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & 0.5 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 11 \\ 10 \end{bmatrix}$$

$y_1 = b_1 = 8$

$y_2 = b_2 - l_{21}y_1 = 11 - (0.5*8) = 7$

$y_3 = b_3 - (l_{31}y_1 + l_{32}y_2) = (0.25*8 + 0.5*7) = 4.5$

The problem of solving a system of linear equations is now reduced to transforming the original coefficient matrix A to upper triangular form. Transformations applied to A must also be be made to the right hand vector b.

Some physical problems which can be reduced to systems of linear equations generate several right hand vectors, b, having the same coefficient matrix, A. Analysis of the transformation process reveals that the operations applied to A, which must also be applied to b, can be delayed without requiring more storage or more operations. This is desirable since order of n^3 ($O(n^3)$) operations are necessary to reduce A to upper triangular form whereas only $O(n^2)$ operations are required to transform b.

A non-singular matrix, A, can be factored into a unit lower triangular matrix, L, and an upper triangular matrix, U.

$$A = LU$$

Substituting LU for A the linear system equation becomes

$$LUx = b.$$

If L were removed from the equation backward substitution could be used to find x, the solution vector. Assuming L is non-singular, L^{-1} can be applied to both sides of the equality producing

$$Ux = L^{-1}b.$$

Finding an inverse to a matrix requires $O(n^3)$ operations and is to be avoided. Let

$$y = L^{-1}b.$$

Premultiplying both sides yields

$$Ly = b.$$

Forward substitution can be used to find y since L is unit lower triangular. Once the original matrix is factored into unit lower and upper triangular forms, solutions to two simple linear systems are required.

$$Ly = b$$

$$Ux = y$$

Vector y must be found first by forward substitution, then x, the solution vector, by backward substitution.

L and U are found by Gaussian elimination. Gaussian elimination transforms the original matrix A into upper triangular form. L is a record of the row operations used to transform A into U. L is created as a by product of Gaussian elimination.

Table 23. Gaussian elimination vector equations

$$r_i = (a_{i1}, a_{i2}, \ldots, a_{in}),\ 1 <= i <= n$$

Column 1

$$m_{k1} = a_{k1}/a_{11},\ 2 <= k <= n$$

$$r_k = r_k - m_{k1}r_k,\ 2 <= k <= n$$

Column 2

$$m_{k2} = a_{k2}/a_{22},\ 3 <= k <= n$$

$$r_k = r_k - m_{k2}r_k,\ 3 <= k <= n$$

. . .

Column n-1

$$m_{n,n-1} = a_{n,n-1}/a_{n-1,n-1}$$

$$r_n = r_n - m_{n,n-1}r_n$$

A matrix is transformed into upper triangular form one column at a time starting with the first column. Row operations are used to generate zeros below the diagonal. In the simplest algorithm pivot rows are chosen starting with row one and continuing to row n-1. A pivot row is successively scaled by multipliers chosen to produce zeros in the column having the same index as the pivot row. Table 23 lists the vector equations to create an upper triangular matrix and Table 24 illustrates the process.

Computations are performed in the order presented in Table 23. All elements in column one, rows two through n, are eliminated in the column one computations. In a similar way all elements in column 2 below the diagonal are eliminated. Finally, in the nth row, only one zero needs to be generated in the n-1 column.

Table 24 illustrates Gaussian elimination. A is transformed into the upper triangular matrix U. Row one is chosen as the pivot row for column one. Multipliers m_{21}, for row two, and m_{31}, for row three, are computed. Vector transformations are applied to rows two and three of matrix A to produce U_1. U_1 is partially triangularized.

Moving to column two, row two of matrix U_1 is chosen as the pivot row and multiplier m_{32} is computed. Only one vector operation is needed to place a zero below the diagonal in column two. The resulting matrix, U_2, is fully triangularized.

It is possible to create a matrix that when used to premultiply U will produce A. That matrix is unit lower triangular and is denoted L. L is formed by placing the multipliers, computed as a by product of Gaussian elimination, in the identity matrix. The subscripts of a multiplier identify the position in the identity matrix where the multiplier is to be inserted. Table 25 shows how

L is constructed in general and Table 26 illustrates how L is created for the example of Table 24.

Table 24. Gaussian elimination example

$$A = \begin{bmatrix} 4 & 1 & 3 \\ 2 & 8 & 1 \\ 1 & 4 & 5 \end{bmatrix}$$

Column 1

$$m_{21} = a_{21}/a_{11} = 0.5$$

$$m_{31} = a_{31}/a_{11} = 0.25$$

$$U_1 = \begin{bmatrix} 4 & 1 & 3 \\ 0 & 7.5 & -0.5 \\ 0 & 3.75 & 4.25 \end{bmatrix} = r_2 = m_{21}r_1 = (2,8,1) - 0.5\ (4,1,3)$$

$$= r_3 = m_{31}r_1 = (1,4,5) - 0.25(4,1,3)$$

Column 2

$$m_{32} = u_{32}/u_{22} = (3.75/7.5) = 0.5$$

$$U_2 = \begin{bmatrix} 4 & 1 & 3 \\ 0 & 7.5 & -0.5 \\ 0 & 0 & 4.5 \end{bmatrix} = r_3 = m_{32}r_2$$

$$= (0,3.75,4.25) - 0.5(0,7.5,-0.5)$$

$$U = U_2$$

Each column of multipliers must be computed after zeros have been generated in the previous column.

The reader can verify that

$$L_iU_i = A,\ 1 <= i <= n-1.$$

In practice L and U are combined in a single n by n matrix. Zeros below the diagonal in an upper triangular matrix and zeros above the diagonal in a unit lower triangular matrix are not needed. The ones along the diagonal of L are assumed. L and U are added together. Table 27 shows the general form of LU factorization and Table 28 illustrates the two matrices combined for the example in Table 26.

Table 25. Unit lower triangular matrix construction

$$j = a_{ij}/a_{jj},\ 2 <= i <= n,\ 1 <= j <= i$$

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ m_{21} & 1 & 0 & \cdots & 0 \\ m_{31} & m_{32} & 1 & \cdots & 0 \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ m_{n1} & m_{n2} & m_{n3} & \cdots & 1 \end{bmatrix}$$

Table 26. Unit lower triangular matrix construction example

$$A = \begin{bmatrix} 4 & 1 & 3 \\ 2 & 8 & 1 \\ 1 & 4 & 5 \end{bmatrix}$$

Column 1

$$m_{21} = a_{21}/a_{11} = 0.5$$

$$m_{31} = a_{31}/a_{11} = 0.25$$

$$L_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & 0 & 1 \end{bmatrix} \qquad U_1 = \begin{bmatrix} 4 & 1 & 3 \\ 0 & 7.5 & -0.5 \\ 0 & 3.75 & 4.25 \end{bmatrix}$$

Column 2

$$m_{32} = u_{32}/u_{22} = 0.5$$

$$L_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & 0.5 & 1 \end{bmatrix} \qquad U_2 = \begin{bmatrix} 4 & 1 & 3 \\ 0 & 7.5 & -0.5 \\ 0 & 0 & 4.5 \end{bmatrix}$$

The element on the diagonal in the pivot row is called the pivot element. Suppose the pivot element is equal to zero. Multipliers cannot be computed because the pivot element is used as the divisor for all the multipliers in that column and zero cannot be used as a divisor. A zero

pivot element implies that row is already triangularized. The solution to this problem is to search the column looking for a non-zero pivot element. If a non-zero pivot element cannot be found, the matrix is singular and no unique solution exists. If the pivot element is found in a row other than the pivot row, the pivot row and the row containing a non-zero pivot element are exchanged. Table 29 illustrates the process.

Table 27. LU combined matrix form

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ m_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ m_{31} & m_{32} & u_{33} & \cdots & u_{3n} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ m_{n1} & m_{n2} & m_{n3} & \cdots & u_{nn} \end{bmatrix}$$

Table 28. LU combined matrix example

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.25 & 0.5 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 4 & 1 & 3 \\ 0 & 7.5 & -0.5 \\ 0 & 0 & 4.5 \end{bmatrix}$$

$$LU = \begin{bmatrix} 4 & 1 & 3 \\ 0.5 & 7.5 & -0.5 \\ 0.25 & 0.5 & 4.5 \end{bmatrix}$$

In Table 29, A is the original matrix. No part of the matrix is triangularized. Multipliers for column one are about to be computed. The pivot element a_{11} in row one is equal to zero. Since $a_{11} = 0$, it cannot be used as the divisor for the multipliers in column one. Column one is searched for the first non-zero element below the diagonal. The first non-zero element in column one is in row two.

Rows one and two are exchanged. Multipliers are computed and LU factorization proceeds.

Table 29. Zero pivot element row exchange example

$$A = \begin{bmatrix} 0 & 5 & 2 \\ 2 & 1 & 5 \\ 3 & 2 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 7 \\ 8 \\ 6 \end{bmatrix}$$

Exchange row 1 and 2 because the pivot element $a_{11} = 0$

$$A = \begin{bmatrix} 2 & 1 & 5 \\ 0 & 5 & 2 \\ 3 & 2 & 1 \end{bmatrix} \quad b = \begin{bmatrix} 8 \\ 7 \\ 6 \end{bmatrix}$$

$$m_{21} = a_{21}/a_{11} = (0/2) = 0$$

$$m_{31} = a_{31}/a_{11} = (3/2) = 1.5$$

A significant point to note is elements of b were interchanged along with the rows of A. A record of the row interchanges must be kept to find y in the matrix equation

$$Ly = b.$$

Elements of b are interchanged according to the record created during LU factorization before successive elements of y are found by forward substitution.

LU factorization has been discussed, largely, without reference to implementation. A straightforward implementation of LU factorization on a computer is not satisfactory because of errors due to finite precision. Large growth in modified matrix elements will cause loss of significant precision. An effective strategy to overcome some problems of finite precision is partial pivoting. Partial pivoting prevents large growth in modified matrix elements. Forsythe and Moler [34] present a theoretical treatment of errors due to finite precision.

The partial pivoting algorithm chooses the pivot row by finding the row having a pivot element with the largest magnitude. The largest pivot element is found by searching the column being triangularized, below the diagonal, for the

element having the largest magnitude. The row having the same index as the column being triangularized and the row having the largest pivot element are exchanged.

4.2 Standard Pascal Transcriptions of DGEFA and DGESL

Original source statements for DGEFA and DGESL can be found in the LINPACK User's Guide [3]. The D is DGEFA and DGESL denotes double precision. The LINPACK User's Guide refers to double precision as full precision meaning real numbers are stored and manipulated as 64-bit values. The GE in DGEFA and DGESL refers to the type of matrix these procedures generate solutions for. GE means general as opposed to positive definite (PO) or triangular (TR). The last two letters indicate the operation being performed. FA indicates DGEFA factors a matrix into unit lower and upper triangular forms. SL signifies that DGESL produces a solution given a factored matrix as input.

Because Pascal is a strongly typed language, arguments to procedures cannot be passed as arguments are passed to subroutines in Fortran. Specifically, Pascal prohibits the use of open array parameters whereas Fortran does not require the array shape to be known at translation-time. As a consequence, Basic Linear Algebra Subroutines (BLAS) DDOT and DAXPY were not used in procedure DGESL. Equivalent inline code was employed instead. In addition BLAS routines needed for DGEFA were simplified, in part, to improve clarity and, in part, because some techniques used in Fortran were impossible to translate to Pascal.

Type declarations for routines described in this section are listed in Table 30. Routines transcribed from the Basic Linear Algebra Subroutine package include DAXPY, in Table 31, DSCAL, listed in Table 32, and IDAMAX, shown in Table 33. Procedure DGEFA, listed in Table 34, calls function IDAMAX and procedures DSCAL and DAXPY to factor the

input matrix A. A listing of DGESL, which finds the solution vector x, is contained in Table 35.

Procedure listings are accompanied by commentary. After the commentary an example is given to illustrate the function of the procedures.

Table 30. Standard Pascal type declarations for DGEFA and DGESL

```
1     TYPE
2       idxrng  = 1..rank;
3       ivector = array[idxrng] of integer;
4       rvector = array[idxrng] of real;
5       matrix  = array[idxrng] or rvector;
```

The declarations in Table 30 are discussed below.

Line 2: Type idxrng is the range over which indexes to arrays defined on the following lines are valid. The constant rank was set to 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50 in separate compilations.

Line 3: Type ivector is an array composed of "rank" integer elements.

Line 4: Type rvector is a real vector containing "rank" elements.

Line 5: Type matrix is an array of rows. Each row is a real vector. There are "rank" number of rows. Type matrix is a "rank" by "rank" array of real elements.

Table 31. Standard Pascal procedure DAXPY

```
1     PROCEDURE daxpy(VAR r,pivotrow: rvector
2                     ;multiplier: real; j: idxrng);
3       VAR c: idxrng;
4     BEGIN
5       FOR c := j TO rank DO
6         r[c] := r[c] + multiplier*pivotrow[c]
7     END{daxpy};
```

Procedure DAXPY, listed in Table 31, performs row elimination. The pivot row is scaled by the multiplier and added to row r below the diagonal. Only elements to the right of the diagonal are manipulated since multipliers belonging to L, the unit lower triangular matrix, are inserted in positions to the left of the diagonal. Variable c is the column subscript. Variable j is the starting position and is equal to one more than the column subscript of the pivot element. Variable r and pivotrow are rows of A, the input matrix to DGEFA.

Table 32 contains a listing of DSCAL which scales a column of the input matrix A. DSCAL is used to produce the multipliers after the pivot element is chosen. The input multiplier is equal to the negative of the reciprocal of the pivot element. Only elements below the diagonal are scaled. Variable c identifies the column and variable i marks the first element in the column to be scaled. Variable r is a local row index.

Table 32. Standard Pascal procedure DSCAL

```
1     PROCEDURE dscal(VAR A: matrix; i,c: idxrng
2                     ;multiplier: real);
3       VAR r: idxrng;
4     BEGIN
5       FOR r := i TO rank DO
6         A[r,c] := multiplier*A[r,c]
7     END{dscal};
```

Function IDAMAX, in Table 33, finds the row index of the pivot element in a column of A. Variable c is the index of the column. The pivot is that element having the largest magnitude. The search for the pivot starts on the diagonal and continues to the end of the column. Variable i is the index of the starting element. Variable pivotrow is the

index of the row containing the pivot element and is returned by function IDAMAX.

Table 33. Standard Pascal function IDAMAX

```
1     FUNCTION idamax(VAR A: matrix; i,c: idxrng): idxrng;
2       VAR t,max: real; pivotrow,r: idxrng;
3     BEGIN
4       max := abs(A[i,c]);
5       pivotrow := i;
6       FOR r := i+1 TO rank DO
7         BEGIN t := abs(A[r,c]);
8           IF t > max THEN
9             BEGIN max := t; pivotrow := r END;
10        END;
11      idamax := pivotrow;
12    END{idamax};
```

Procedure DGEFA, listed in Table 34, factors the input matrix A into a unit lower triangular matrix, L, and an upper triangular matrix, U. L and U are combined in a single matrix and replace A. Input variable lda is a vestige of the Fortran version of DGEFA and is not used in this version. The value of the variable n is the order of the matrix A. Variable ipvt is a record of the row interchanges which occur as a result of partial pivoting. info is normally set to zero. However, if a non-zero pivot element cannot be found, info will be set to the index of that column.

Lines 7,8, and 9: Variable multiplier is the negative reciprocal of the pivot element. Variable t is used to interchange elements of rows. Variable r is the row index and variable c is the column index. Variable k is used as the column index when rows are exchanged.

Line 11: Assume all pivot elements will be non-zero.

Line 12: Gaussian elimination is performed one column at a time starting with the first column. All columns to

the left of the diagonal are processed in sequence. The nth column, the column on the diagonal, is not processed.

Table 34. Standard Pascal procedure DGEFA

```
1     PROCEDURE dgefa (VAR A     : matrix
2                      ;     lda   : idxrng
3                      ;VAR n     : idxrng
4                      ;     ipvt  : ivector
5                      ;VAR info  : integer
6                      );
7       VAR
8         multiplier,t:   real;
9         r,c,k,pivotrow: idxrng;
10      BEGIN{dgefa}
11        info := 0;
12        FOR c:=1 TO rank-1 DO
13          BEGIN r := c;
14            pivotrow := idamax(A,r,c);
15            ipvt[r] := pivotrow;
16            IF abs(A[pivotrow,c]) < smallreal then
17              info := c
18            ELSE
19              BEGIN
20                IF pivotrow <> r then
21                  FOR k := c TO rank DO
22                    BEGIN
23                      t := A[r,k];
24                      A[r,k] := A[pivotrow,k];
25                      A[pivotrow,k] := t
26                    END;
27                pivotrow := r;
28                multiplier := -1.0/A[pivotrow,c];
29                dscal(A,pivotrow+1,c,multiplier);
30                FOR r := pivotrow+1 TO rank DO
31                  daxpy(A[r],A[pivotrow],A[r,c],c+1);
32              END;
33          END;
34        IF abs(A[n,n]) < smallreal then info := n;
35        ipvt[rank] := rank;
36      END{dgefa};
```

Line 13: Assume the pivot row index is equal to the column index.

Lines 14 and 15: Search column c, below the diagonal, for the row index of the pivot element. Record the row in which the pivot occurs in ipvt.

Lines 16 and 17: Ensure the pivot has magnitude greater than zero. Record the column having a pivot element equal to zero in info.

Lines 20 through 26: Interchange rows to the right of the diagonal if row r and the pivot row are not identical.

Line 27: After (possibly) interchanging rows the pivot row index is equal to the current row.

Lines 28 and 29: Compute multipliers for column c.

Lines 30 and 31: Perform row elimination on all rows below the pivot row and to the right of the diagonal.

Line 34: Record a zero pivot for column n.

Line 35: Complete the row interchange record.

Table 35 contains a listing of DGESL which first solves the matrix equation

$$Ly = b$$

for y using forward substitution and then solves the matrix equation

$$Ux = y$$

for x using backward substitution. L and U are combined in the input matrix A. Array ipvt contains a record of row interchanges. Variables lda and job are not used in this implementation.

Lines 12 through 23: Forward substitution is used to solve the equation

$$Ly = b$$

for y. The input vector, b, is transformed to y one element at a time.

Lines 15 through 20: Elements of b are interchanged according to the row interchange record in array ipvt.

Lines 21 and 22: L contains the negative of the multipliers required in the equation

$$LU = A.$$

Thus, the expression in line 22 is a sum rather than a difference as previously discussed. As elements of b are interchanged they are applied in all relevant solutions to elements of y. Recall that y replaces b and in Table 35 y is the vector b.

Table 35. Standard Pascal procedure DGESL

```
1     PROCEDURE dgesl (VAR A    : matrix
2                     ;     lda : idxrng
3                     ;     n   : idxrng
4                     ;VAR ipvt: ivector
5                     ;VAR b    : rvector
6                     ;     job : integer
7                     );
8       VAR
9         r,c,i,pivotrow: idxrng;
10        t             : real;
11    BEGIN{dgesl}
12      FOR r := 1 TO rank DO
13        BEGIN
14          pivotrow := ipvt[r];
15          IF r <> pivotrow THEN
16            BEGIN
17              t := b[pivotrow];
18              b[pivotrow] := b[r];
19              b[r] := t
20            END;
21          FOR c := r+1 TO rank do
22            b[c] := b[c] + A[c,r]*b[r];
23       END;
24    FOR r := rank DOWNTO 1 DO
25      BEGIN
26        t := b[r];
27        FOR c := r+1 TO rank DO
28          t := t - A[r,c]*b[c];
29        b[r] := t/A[r,r];
30      END;
31    END{dgesl};
```

Lines 24 through 30: Backward substitution is used in the equation

$$Ux = y$$

to find x. The b-vector contains y and is transformed into x by statements on lines 24 through 30.

Table 36 illustrates how procedure DGEFA factors a matrix. Matrix A is listed in the column titled "Matrix Transformations." Matrix A is transformed from its original form to LU form. The column titled "ipvt" shows the creation of the row interchange record during factorization. The variable multiplier is the negative of the reciprocal of the pivot. This quantity is shown in the column titled "Multiplier." The variable multiplier should not be confused with multipliers that are created as a result of LU factorization. The factorization multipliers are in the unit lower triangular matrix below the diagonal. The column titled "Comments" briefly indicates the action being performed. Line numbers refer to the line numbers in Table 34. Statements associated with the line numbers cause the transformations illustrated in Table 36.

Seven matrix transformations are shown in Table 36. The first is the initial matrix to be factored. Transformation two, three, and four apply to column one and transformations five, six, and seven apply to column two.

Transformation two illustrates the search for the pivot element. The pivot is that element having the largest magnitude. All elements in column one are tested. Row three contains the pivot. Rows one and three are exchanged. Row three is recorded in the first element of array ipvt, the row interchange record. Variable multiplier, the scale factor input to DSCAL, is computed by producing the negative reciprocal of the pivot.

The multipliers in column one, belonging to L, are shown in the third transformation.

Row one, scaled by the multiplier in row two, is added to row two in transformation four. The vector sum replaces the original row two. Only elements to the right of column

one are affected. In a similar way row one is scaled by the multiplier in row three and added to row three. The sum replaces the original row.

Table 36. Example execution of DGEFA

Matrix Transformations	ipvt	multiplier	Comment	Line Number
$\begin{bmatrix} 2 & 8 & 1 \\ 1 & 4 & 5 \\ 4 & 1 & 3 \end{bmatrix}$			Initial Matrix	13-28
$\begin{bmatrix} 4 & 1 & 3 \\ 1 & 4 & 5 \\ 2 & 8 & 1 \end{bmatrix}$	$\begin{bmatrix} 3 \\ X \\ X \end{bmatrix}$	-0.25	Exchange rows 1 and 3	29
$\begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 4 & 5 \\ -0.5 & 8 & 1 \end{bmatrix}$			Compute multipliers for col. 1	30-31
$\begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 3.75 & 4.25 \\ -0.5 & 7.5 & -0.5 \end{bmatrix}$			Row elimination for col. 1	30-31
$\begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 7.5 & -0.5 \\ -0.5 & 3.75 & 4.25 \end{bmatrix}$	$\begin{bmatrix} 3 \\ 3 \\ X \end{bmatrix}$	-0.27	Exchange rows 2 and 3	13-28
$\begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 7.5 & -0.5 \\ -0.5 & -0.5 & 4.25 \end{bmatrix}$			Compute the multiplier for col. 2	29
$\begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 7.5 & -0.5 \\ -0.5 & -0.5 & 4.5 \end{bmatrix}$			Row elimination for col. 2	30-31

Since row three contains the largest element in column two, rows two and three are exchanged in transformation five. A record of the exchange is stored in the second element of array ipvt. Rows are exchanged in columns two and three only.

Transformation six shows the unit lower triangular matrix multiplier in row three, column two.

Transformation seven shows row elimination for column two. Only the element in row three, column three is altered. The matrix is in LU form.

Table 37 illustrates how procedure DGESL solves the matrix equation

$$Ly = b$$

for y using forward substitution. Line numbers are listed beside the statements which transform b into y in Table 37. The input matrix A is shown at the top of Table 37 in LU form along with the row interchange record, array ipvt.

Table 37. DEGSL example of Ly = b

Line Numbers	Comments	b
	$A = \begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 7.5 & -0.5 \\ -0.5 & -0.5 & 4.5 \end{bmatrix}$ $ipvt = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}$ $b = \begin{bmatrix} 11 \\ 10 \\ 8 \end{bmatrix}$	
14-19 14-19 14-19	exchange rows one and three	$\begin{bmatrix} 8 \\ 10 \\ 11 \end{bmatrix}$
20-21 20-21 20-21	b[2] := b[2] + A[2,1]*b[1] = 10 - 0.25*8 = b[3] := b[3] + A[3,1]*b[1] = 11 - 0.5 *8 =	$\begin{bmatrix} 8 \\ 8 \\ 7 \end{bmatrix}$
14-19 14-19 14-19	exchange rows two and three	$\begin{bmatrix} 8 \\ 7 \\ 8 \end{bmatrix}$
20-21 20-21 20-21	b[3] := b[3] + A[3,2]*b[2] = 8 - 0.5 *7	$\begin{bmatrix} 8 \\ 7 \\ 4.5 \end{bmatrix}$

Five instances of vector b are shown Table 37. The first instance is vector b as it is input to procedure DGESL. Vector b has been transformed into vector y in the last instance.

The second and third instance of vector b illustrate the solution to y_2 and the fourth and fifth instance of vector b depict the solution to y_3.

Table 38 illustrates how procedure DGESL solves the equation

$$Ux = y$$

for x using forward substitution. Input matrix A and vector y are shown at the top of the Table. Vector y is contained in array b. Vector y is transformed into x in this part of procedure DGESL.

Table 38. DEGSL example of Ux = y

Line Numbers	Comments
	$A = \begin{bmatrix} 4 & 1 & 3 \\ -0.25 & 7.5 & -0.5 \\ -0.5 & -0.5 & 4.5 \end{bmatrix}$ $b = \begin{bmatrix} 8 \\ 7 \\ 4.5 \end{bmatrix}$
26	t := b[3] = 4.5
27,28	--
29	b[3] := t/A[3,3] = 4.5/4.5 = 1
26	t := b[2] = 7
27,28	t := t - A[2,3]*b[3] = 7 + 0.5*1 = 7.5
29	b[2] := t/A[2,2] = 7.5/7.5 = 1
26	t := b[1] = 8
27,28	t := t - A[1,2]*b[2] - A[1,3]*b[3]
27,28	= 8 - 1*1 - 3*1 = 4
29	b[1] := t/A[1,1] = 4/4 = 1

Three sets of statements are shown in Table 38. The first set illustrates the solution of x_1. The second and third sets show computations for x_2 and x_3. Vector x is contained in array b.

4.3 Vector Pascal Transcriptions of DGEFA and DGESL

The purpose of this section is to explain alterations to procedures DGEFA and DGESL made to exploit the vector operations available in Vector Pascal. Vector operations make procedures DAXPY and DSCAL unnecessary and they have been replaced by equivalent vector expressions. Function IMAX, which replaces IDAMAX, illustrates the use of an open array parameter. Both procedure DGEFA and DGESL employ vector slices.

Table 39 lists type declarations for procedures DGEFA, DGESL, and function IMAX. Table 40 contains a listing of function IMAX. Procedures DGEFA and DGESL are listed in Tables 41 and 42 respectively. Commentary accompanies procedure listings.

Table 39. Vector Pascal type declarations for DGEFA and DGESL

```
1     TYPE
2       idxrng  = 1..rank;
3       ivector = array[idxrng] of integer;
4       rvector = array[idxrng] of real;
5       matrix  = array[idxrng] or rvector;
6     rvec    = array of real
```

The declarations in Table 39 are identical to those in Table 31 on lines one through five. Line 6 defines an open array parameter used in function IMAX. Type rvec is an array of real numbers whose bounds are dynamically altered during execution.

Table 40. Vector Pascal function IMAX

```
1     FUNCTION imax(v: rvec): idxrng;
2       VAR r,l,maxi: idxrng; max,t: real;
3     BEGIN l := low(v)
4       max := abs(v[l]); maxi := l;
5       FOR r := l+1 TO high(v) DO
6         BEGIN t := abs(v[r]);
7           IF t > max THEN
8             BEGIN max := t; maxi := r END;
9         END;
10      imax := maxi;
11    END{imax};
```

Function IMAX finds the row index of the pivot element in the input column-slice of A. Function IMAX replaces IDAMAX found in Table 33. The essential difference between IMAX and IDAMAX is the way a column-slice of matrix A is

passed. Function IDAMAX receives, as separate parameters, the address of A, the index of the column to be searched, and index of the row where the search begins. In contrast, function IMAX receives an open array parameter which implicitly describes the column of matrix A to search.

Array v is the column slice of the matrix A and is an open array parameter. The pivot is that element having the largest magnitude. The search for the pivot proceeds sequentially from the lowest numbered index of array v to the highest numbered index. Standard functions low and high extract the low and high bounds respectively. Variable max is equal to the absolute value of the element having the largest magnitude. Variable maxi holds the index of the element with the largest magnitude.

Recall procedure DGEFA factors input matrix A into a unit lower triangular matrix L and an upper triangular matrix U. Both L an U are combined into matrix A during execution of DGEFA. Table 36 illustrates the intermediate forms of matrix A as it is factored.

The following paragraphs describe the differences between the Standard Pascal version of procedure DGEFA as shown in Table 34 and the Vector Pascal version listed in Table 41.

Variables t and k do not appear in the Vector Pascal version of procedure DGEFA. Both variables are used to exchange rows in the Standard Pascal version. Vector Pascal employs standard function swap to exchange rows. Line 21 in Table 41 shows how function swap is used. Line 21 in Table 41 replaces lines 21 through 26 in Table 34.

Table 41. Vector Pascal Procedure DGEFA

```
1     PROCEDURE dgefa (VAR A    : matrix
2                      ;    lda  : idxrng
3                      ;VAR n    : idxrng
4                      ;    ipvt : ivector
5                      ;VAR info : integer
6                      );
7       VAR
8         multiplier  : real;
9         r,c,pivotrow: idxrng;
10      BEGIN{dgefa}
11        info := 0;
12        FOR c:=1 TO rank-1 DO
13          BEGIN r := c;
14            pivotrow := imax(A[r..rank,c]);
15            ipvt[r] := pivotrow;
16            IF abs(A[pivotrow,c]) < smallreal then
17              info := c
18            ELSE
19              BEGIN
20                IF pivotrow <> r then
21                  swap(A[r,c..rank],A[pivotrow,c..rank]);
22                pivotrow := r;
23                multiplier := -1.0/A[pivotrow,c];
24                A[pivotrow+1..rank,c] :=
25                  multiplier*A[pivotrow+1..rank,c];
26                FOR r := pivotrow+1 TO rank DO
27                  A[r,c+1..rank] := A[r,c+1..rank] +
28                    A[r,c]*A[pivotrow+1..rank,c]
29              END;
30          END;
31        IF abs(A[n,n]) < smallreal then info := n;
32        ipvt[rank] := rank;
33      END{dgefa};
```

Function IMAX is called instead of IDAMAX in the Vector Pascal version of procedure DGEFA. Function IMAX accepts a vector as an argument whereas IDAMAX requires three arguments to describe the column of matrix A to search for the pivot. The vector passed to IMAX is a slice of column c beginning on the diagonal and extending to the last row. Function IMAX is called on line 14 in Table 41.

Procedure DSCAL is used to compute the multipliers for column c in the Standard Pascal version of procedure DGEFA.

Multipliers are computed in line 29 of Table 34. Vector operations can be used to perform the function of DSCAL in Vector Pascal. Lines 24 and 25 in Table 41 depict a vector statement equivalent to procedure DSCAL as it is employed in the Standard Pascal version of DGEFA. Variable multiplier is used to scale a column-slice of matrix A (line 25). The resulting vector replaces the original vector (line 24). The column-slice begins one row below the diagonal and continues to the last row of the matrix. The index of the column-slice is c.

Procedure DAXPY, shown in Table 34, line 31, is replaced by the vector assignment on lines 27 and 28 of Table 41. The vector assignment replaces a row-slice of matrix A with the sum of the original vector plus a scaled version of the pivot row. All row-vectors include elements to the right of the diagonal. The pivot row is scaled by that factor computed to produce a zero in column c when the scaled pivot row is added to row r.

Procedure DGESL in Table 42 first solves the matrix equation

$$Ly = b$$

for y and then solves the matrix equation

$$Ux = y$$

for x. Forward substitution is used to solve

$$Ly = b$$

and backward substitution is used to solve

$$Ux = y.$$

The example in Table 37 shows how procedure DGESL transforms array b from its original form to vector y, the desired outcome of forward substitution. Table 38 illustrates how procedure DGESL finds vector x using backward substitution.

The paragraphs below describe the differences between the Vector Pascal implementation of procedure DGESL shown in Table 42 and the Standard Pascal version listed in Table 35.

Table 42. Vector Pascal procedure DGESL

```
1     PROCEDURE dgesl (VAR A    : matrix
2                     ;     lda : idxrng
3                     ;     n   : idxrng
4                     ;VAR ipvt: ivector
5                     ;VAR b    : rvector
6                     ;     job : integer
7                     );
8       VAR
9         r,c,i,pivotrow: idxrng;
10        t             : real;
11    BEGIN{dgesl}
12      FOR r := 1 TO rank-1 DO
13        BEGIN
14          pivotrow := ipvt[r];
15          IF r <> pivotrow THEN
16            BEGIN
17              t := b[pivotrow];
18              b[pivotrow] := b[r];
19              b[r] := t
20            END;
21          c := r+1;
22          b[c..rank] := b[c..rank] + A[c..rank,r]*b[r];
23        END;
24      b[rank] := b[rank]/A[rank,rank];
25      FOR r := rank-1 DOWNTO 1 DO
26        b[r] := (b[r] -
27                 A[r,r+1..rank]*b[r+1..rank])/A[r,r];
28    END{dgesl};
```

Lines 12 through 23 of procedure DGESL in Table 42 implement the forward substitution algorithm and lines 24 through 27 implement backward substitution.

The for-loop on lines 21 and 22 in the Standard Pascal version of DGESL shown in Table 35 is replaced by a single vector statement in the Vector Pascal version shown on line 22 of Table 42. Two vectors are added and the sum is used to replace vector b. The plus sign in the expression on line 22 of Table 35 denotes vector addition rather than scalar addition. The vector to the right of the plus sign is a column-slice of matrix A scaled by the rth element of

vector b. All three vectors in the statement extend from row c to the last row (rank).

Line 24 in Table 42 shows the straightforward solution of the first equation in the backward substitution algorithm. Lines 25 through 27 show how Vector Pascal can be used to find the solution to the remainder of the backward substitution equations. The expression inside the parentheses is a scalar. The scalar is the difference between an element of vector b and the inner product of two vectors. The asterisk on line 27 indicates that the inner product operation is performed. The vector to the left of the asterisk on line 27 is a row-slice of matrix a beginning in column r+1 and continuing to the last column of A. The vector to the left of the asterisk includes all elements of vector b having an index greater than r.

5. VECTOR PASCAL IMPLEMENTATION

This chapter describes implementation details of the Vector Pascal compiler. The process a programmer would use to translate and execute Vector Pascal programs is described in section 5.1. Vector Pascal statements are translated to P-code instructions. A hypothetical computer called a P-machine interprets P-code instructions. Section 5.2 discusses the P-machine architecture. Alterations were made to the P-code instruction set to accommodate compilation and to facilitate vector operations. Alterations to the P-code instruction set are discussed in section 5.3. P-code instructions are translated to Array Processor Assembler Language (APAL) statements by the P-code to APAL Assembler. A detailed discussion of the assembler is beyond the scope of this dissertation. However, an example is presented in section 5.4 to illustrate the techniques used to translate P-code instructions to APAL statements.

5.1 Component Overview

To translate Vector Pascal programs to executable form requires four translation steps. The first two steps, translation to P-code and translation to Array Processor Assembly Language (APAL), occur on a Personal Computer AT. The latter two translation steps, translation to relocatable object and binding, occur on an IBM 308X computer executing under control of its operating system VM/CMS. Vector Pascal programs execute on a FPS-164 Scientific Computer and under the control of Single User Monitor (SUM), the operating system of the FPS-164 Scientific Computer. SUM is supervised by System Job Executive (SJE) which executes on the host IBM 308X computer. Designers of Vector Pascal programs initiate execution of their programs using facilities of the SJE. Figure 5 depicts the process by which Vector Pascal programs are translated and executed.

The translation process is explained by example. Assuming the DOS file lintest.pas contains a Vector Pascal program, the following paragraphs describe each translation step. A summary of the translation steps is contained in Table 43.

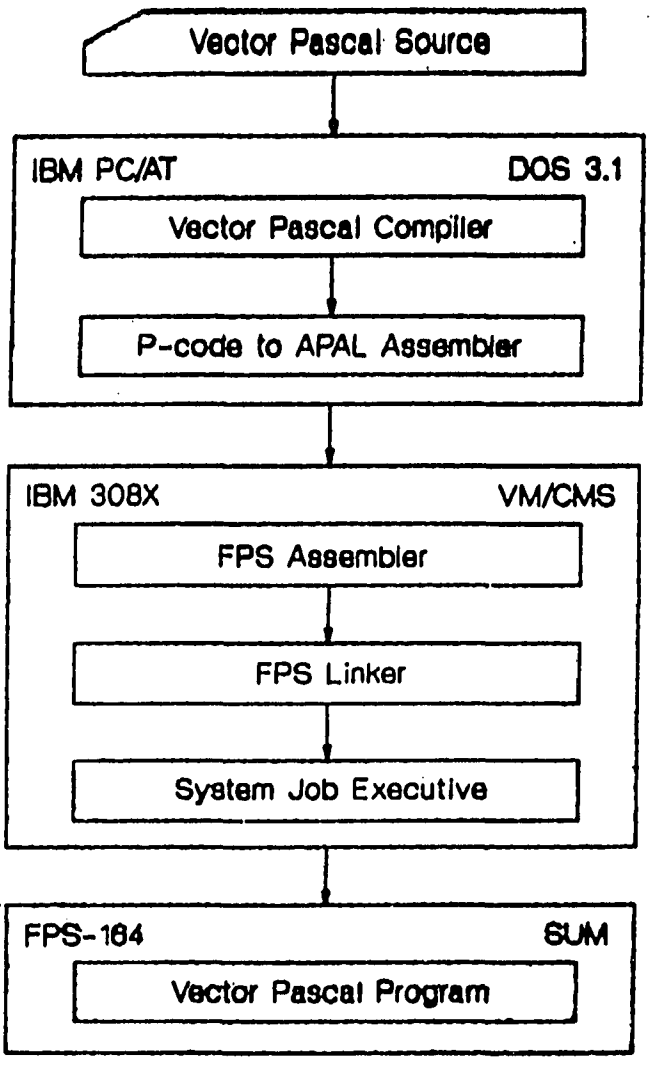


Figure 5. Vector Pascal translation

The compiler reads a text file containing Vector Pascal source on an IBM Personal Computer AT. The compiler executes under the control of the Disk Operating System

(DOS) version 3.1. The Vector Pascal compiler is invoked by entering

```
c4 lintest
```

on the DOS command line. The identifier c4 is the name DOS 3.1 uses to invoke the Vector Pascal compiler. The name lintest refers to the source file containing a Vector Pascal program. The Vector Pascal compiler assumes that the DOS file extension to lintest is pas. The complete file name of the source file in this example is lintest.pas.

The Vector Pascal compiler translates a source program into an equivalent P-code program. Section 5.2 describes details of the P-code Machine.

Table 43. Instructions for compiling and executing LINTEST

Command	Description
On a Personal Computer AT	
c4 lintest	Translate Vector Pascal to P-code
a4 lintest	Translate P-code to APAL
Copy DOS file lintest.asm to VM/CMS file LINTEST APAL64 A	
On an IBM 308X with VM/CMS	
APAL64 LINTEST (DIAG(ERR) LIST	Translate APAL to relocatable object
APLINK64 LINTEST,PASCALIO (LIST	Bind LINTEST to Vector Pascal run-time library
SJE	Enter System Job Executive
COPYIN/B 'LINTEST APIMG64 A',LINTEST	Copy VM/CMS executable file to FPS file
LINTEST	Execute Vector Pascal program LINTEST

The name of the text file containing the P-code program is the same as the source program but having a DOS file

extension of prr. In the example above the file lintest.prr would be created as a result of submitting the command:

```
c4 lintest
```

P-code is translated to Array Processor Assembly Language by the P-code to APAL Assembler. The Assembler is known to DOS as a4. The P-code program lintest can be assembled by entering the command:

```
a4 lintest
```

The name of the text file produced by the P-code to APAL Assembler is the same as the P-code source file name but having a DOS file extension of asm. In the example above the file lintest.asm would be created as a result of submitting the command:

```
a4 lintest
```

The result of executing the Vector Pascal compiler and P-code to APAL Assembler is an APAL program equivalent to the original Vector Pascal source.

Two more translation steps are required before the program can be executed. These steps are executed on an IBM 308X computer running under the control of the VM/CMS operating system. The APAL program is copied to diskette and the diskette is taken to a Personal Computer AT that is connected to an IBM 308X computer. The file containing the APAL program is subsequently copied to a VM/CMS file having the same file name as the source file and a file type of APAL64. In the example above the DOS file lintest.asm is copied to the VM/CMS file LINTEST APAL64 A.

VM/CMS file identifiers consist of a file name, a file type, and a file mode. The file mode denotes which mini-disk a file belongs to. In the example, LINTEST APAL64 A, A is the file mode. The file identifier must have a file mode but no particular requirements are imposed on the file mode of the APAL program file.

Once the APAL program is in a VM/CMS file it can be translated to relocatable object form. The FPS Assembler, which can be purchased from Floating Point Systems (FPS), Inc. performs this task. The APAL program in VM/CMS file LINTEST APAL64 A can be assembled by entering

```
APAL64 LINTEST (LIST DIAG(ERR)
```

on the VM/CMS command line. APAL64 is the name by which the VM/CMS operating system knows the FPS Assembler. The LIST option following the parentheses directs the FPS Assembler to produce a listing of the APAL source statements beside the hexadecimal representation of the relocatable object program. The DIAG(ERR) option directs the FPS Assembler to suppress certain warnings which are issued as a result of naming conventions adopted by the P-code assembler. The FPS Assembler produces a file having the same file name and file mode as the input source and a file type of APTXT64. In the example, the FPS Assembler, would create files LINTEST APTXT64 A and LINTEST LISTING A. The APTXT64 file contains relocatable object code and the LISTING file contains APAL statements beside the hexadecimal encoding of the symbolic form of the statements.

The last step in translation is binding. The FPS Linker, known to VM/CMS as APLINK64, binds the relocatable object file produced in the previous step to routines in the Vector Pascal run-time library. An executable image is created the FPS Linker. The program LINTEST is bound by entering

```
APLINK64 LINTEST,PASCALIO (LIST
```

on the VM/CMS command line. Files LINTEST APIMG64 A and LINTEST APMAP64 A are created as a result of the command. The APIMG64 file contains an executable image of the LINTEST program. The APMAP64 file describes all external symbols in terms of their lengths and absolute addresses. File

PASCALIO APLIB64 A is the run-time library and contains Vector Pascal standard procedures and functions.

System Job Executive enables the user to execute programs on a FPS-164. The FPS-164 Scientific Computer is connected to an IBM 308X host computer via an IBM Block Multiplexer Channel. SJE executes on the host computer and supervises the operating system on the FPS-164. Single User Monitor manages the resources of the Scientific Computer under the direction of SJE. Communication between SJE and SUM is accomplished via channel programs.

SJE accepts interactive commands permitting the user to control the resources of the Scientific Computer. SJE is invoked by typing

```
SJE
```

on the VM/CMS command line. A program must be copied to the Scientific Computer before it can be executed. Assuming the executable VM/CMS file LINTEST APIMG64 A exists and SJE is executing, the command

```
COPYIN/B 'LINTEST APIMG64 A',LINTEST
```

can be issued. The effect of the command is to copy the executable file enclosed in apostrophes to an FPS file named LINTEST.

After the executable has been copied to the Scientific Computer it can be executed. Assuming FPS file LINTEST exists it can be executed by entering the following SJE command:

```
LINTEST
```

5.2 The P-machine

The purpose of this section is to describe the instruction set architecture of the P-machine, a hypothetical computer that interprets P-codes. Discussion and diagrams which appear in this section are taken from Pascal Implementation: The P4 Compiler [3]. An explanation

of the P-machine execution-time memory management precedes the definition of the P-code instruction set.

Five registers enable the P-machine to define the data and execution environment. The registers are:

i PC, the program counter
ii SP, the stack pointer
iii MP, the mark stack pointer
iv NP, the new pointer
v EP, the extreme stack pointer

The program counter, PC, contains the address of the next instruction to be executed. The remainder of the registers are used to manage memory.

Memory is divided into three areas as shown in Figure 6. Constants are assigned addresses starting at the largest memory address and continuing toward the lowest memory address. The stack starts at the lowest memory address and grows toward the heap. The stack pointer, SP, points to the address of the first unused location in memory above the stack. Each time a procedure or function is called a new stack frame is created on the stack. Figure 7 illustrates the organization of a stack with three active procedures or functions. Each time a procedure or function completes the most recently allocated stack frame is returned to free memory.

The heap starts at the bottom of the constants and grows toward the free area. Standard procedures new and

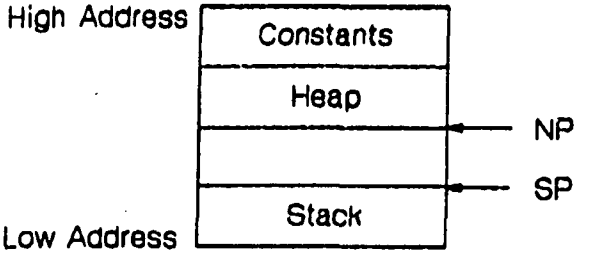


Figure 6. Standard Pascal memory organization

dispose allocate and return records to the heap. The new pointer, NP, points to the first memory location below the heap. If the stack pointer ever becomes greater than the new pointer the executing program has consumed more memory than is available. This event is called a collision. The program is terminated abnormally.

Figure 8 depicts the organization of a stack frame. The stack mark enables procedures and functions to access data outside their local variables and parameters, and return to their calling procedures. Parameters are pushed on the stack by the calling procedure or function. Local variables are allocated above parameters. The arithmetic stack contains intermediate values belonging to the current statement.

The mark pointer, MP, points to the base of the stack frame. All references to values within the stack frame are made relative to the mark pointer.

The extreme pointer, EP, points to the top of the current stack frame. The stack pointer will not exceed the

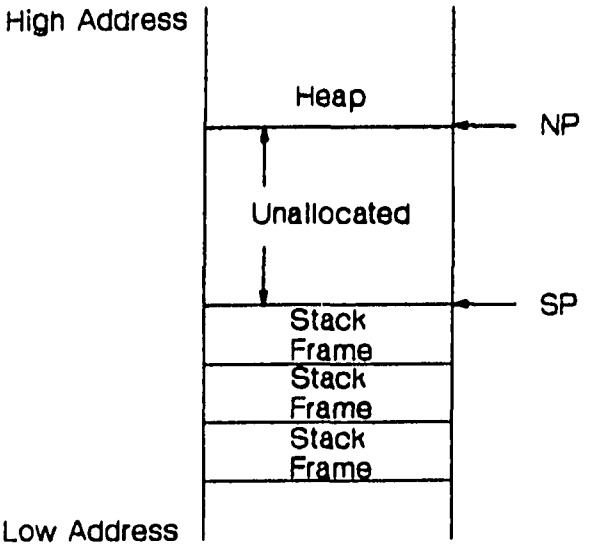


Figure 7. Standard Pascal stack organization

extreme pointer during the execution of a procedure or function. The extreme pointer allows the P-machine to increment the stack pointer without fear of collision. The maximum possible size of each stack frame is known at compile time and this is reflected by the value of EP.

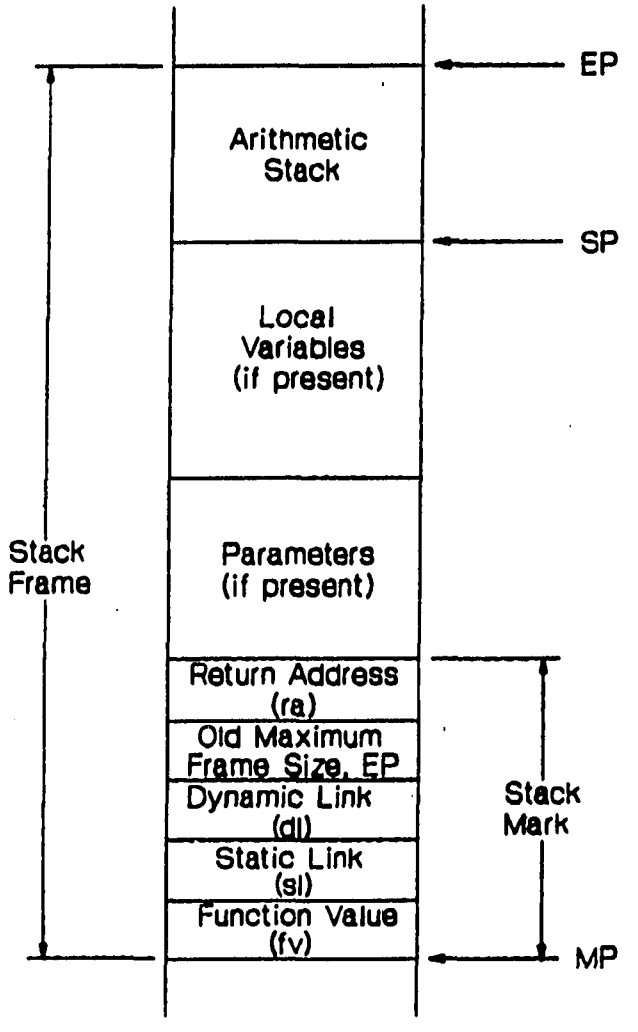


Figure 8. Standard Pascal stack frame

The Standard Pascal program in Table 44 followed by corresponding P-codes in Table 45 illustrate how stack frames are created and used. Figure 9 depicts the stack

produced by program E1 after the P-code instruction on line 7 in Table 45 has been executed.

Program E1 in Table 44 contains one internal procedure, p, which increments its parameter, i, by one. Parameter i is passed by reference, meaning the address of j is passed to procedure p when p is called on line 9. Global variable j is initialized on line 8. When procedure p returns after line 9, variable j is set to two.

P-codes in Table 45 have the form

operand C P Q

The operand is a three-letter mnemonic. For example, mst is a mnemonic for mark stack. Selected instructions have a one letter type parameter that is directly appended to the operand. This parameter, C, denotes one of the primitive types shown in Table 46. Definition of the P and Q parameters are dependent on the P-codes which use them.

Table 44. Standard Pascal program E1

Line	Pascal Statement	Description
1	program E1(input,output);	Program E1 starts here
2	var j: integer;	Variable j is global
3	procedure p(var i:integer);	Procedure p is internal
4	begin{p}	to program E1. Parameter i is passed by address.
5	i := i + 1;	Increment parameter i.
6	end{p};	Terminate procedure p
7	begin{E1}	Begin the body of E1
8	j := 1;	Initialize variable j.
9	p(j);	Call procedure p. Variable j is incremented.
10	end{E1}.	Terminate program E1

Table 45. P-codes for program E1

Line	P-code				Description
1	1 3				Procedure p begins here.
2	ent	1	1	4	14 is set to the size of the p stack frame less the arithmetic stack
3	ent	2	1	5	15 is set to the size of the p stack frame
4	loda	0		5	Push the address of i on the stack, to be consumed by stoi on line 9.
5	loda	0		5	Push the address of i on the stack, to be consumed by indi on line 6.
6	indi			0	Fetch the value of i
7	ldci		1		Push constant 1
8	adi				Add top two items on stack, i and 1.
9	stoi				Store sum on top of the stack in the address next to the top of the stack
10	retp				Return to caller, E1
11	1 4=			6	Stack mark(5)+locals(0)+
12	1 5=			8	Stack mark(5)+locals(0)+ parameters(1)+arithmetic stack(3) - 1.
13	1 6				Entry point for body of E1.
14	ent	1	1	7	Points to stack frame size less arithmetic stack for E1.
15	ent	2	1	8	Points to stack frame size for E1.
16	ldci		1		Push constant 1
17	sroi			9	Store value on top of stack in j.
18	mst			0	Mark stack, prepare to call p.
19	lao			9	Push address of j, the only argument of p, on the stack
20	cup	1	1	3	Call procedure p.
21	retp				Return to initialization

Table 45. Continued

Line	P-code				Description
22	1	7=		10	Base stack mark(5)+locals (1)+parameters(0)+ base IO(4)
23	1	8=		6	Base stack mark(5)+locals (1)+arithmetic stack(1) - 1
24	mst			0	Construct base stack mark
25	cup	0	1	6	Call body of E1
26	stp				Stop

Execution of the P-code program in Table 45 begins on line 22. The body of program E1 starts on line 13 and extends to line 23. Procedure p begins on line 1 and continues through line 12. Each line of the P-code program is explained in the following paragraphs. Lines are explained in the order that they are executed from first to last.

Table 46. P-code primitive types

Mnemonic	Primitive Type
a	Address
b	Boolean
c	Character
i	Integer
r	Real
s	Set
x	Any of the above types

Line 22: The base stack mark is created by the first instruction executed, mark stack (mst). The operand of the mst instruction is an indication of the depth of nesting of the given procedure. The depth is defined as "one plus the level of the calling procedure minus the level of the called procedure." Execution of the mst instruction creates values for the static and dynamic links, and saves the current extreme pointer. The dynamic pointer always points to the

base of the previous stack frame. The static pointer points to the stack frame of the procedure or function lexically enclosing the current procedure.

Line 23: The call user procedure (cup) instruction directs the P-machine to call the main body of program E1 which starts at label 1 6 on line 13. The first operand of the cup instruction is the number of arguments that are passed to the procedure or function which starts at the label in the second operand. The cup instruction places the return address in the stack mark, updates the mark pointer, MP, and transfers control to the label.

Lines 14 and 15: Entry (ent) instructions precede the main body of every procedure or function. The entry instruction has two operands P and Q. The second operand is a label. For example, the entry instruction on line 13 has 1 7 in the Q operand position. The first operand is either a 1 or a 2. If the first operand is equal to 1, then the label is set to the size of the stack frame less the arithmetic stack. For example, label, 1 7, in the entry instruction on line 14, is set to 10, indicating that the stack frame less the arithmetic stack for program E1 is equal to 10. The reader can verify the stack frame size for program E1 in Figure 9. If the first operand of an entry instruction is equal to 2 as it is on line 15 of Table 45, then the label is set to the value taken on by the extreme pointer, EP.

Lines 16 and 17: The body of program E1 continues after the entry instructions on lines 14 and 15 of Table 45. The Pascal statement on line 8 of Table 44 is translated into instructions on line 16 and 17 which initialize the variable j. The address of j (9) is the second operand of the store at base-level address (sro) instruction. Variable j's position in the stack can be seen in Figure 9.

Line 18: The stack mark for procedure p is created.

Line 19: The address of variable j is pushed on the stack directly above the stack mark for procedure p by the load base-level address (lao) instruction. The address of variable j is the parameter for procedure p (see Figure 9).

Line 20: Procedure p is invoked at label 1 3 with one argument.

Lines 1 through 3: Procedure p starts on line 1. The entry instruction on line 2 gives the size of the stack mark, local variables, and parameters. Figure 9 shows five memory locations allocated to the stack mark for procedure p, no memory allocated for local variables, and one location allocated for the parameter of procedure p. The total is six and label 1 4 referenced in the entry statement on line 2 is set to 6 (see line 11). The entry statement on line 3 sets the extreme pointer.

Lines 4 through 9: The assignment statement on line 5 of Table 44 is translated into the P-codes on lines 4 through 9 of Table 45. Variable i receives the value of the expression on the right hand side of line 5, Table 44. The address of variable i is required for the assignment to occur. The assignment is completed on line 17 of Table 45. The address of variable i is pushed on the stack by the load (lod) instruction on line 4, Table 45. The character a following the lod instruction indicates that an address is being placed on the stack. Recall that variable i is the argument of procedure p and is actually the address of variable j. The lod instruction on line 4 actually pushes the address of variable j on the stack.

Lines 5 and 6: The lod instruction on line 5 places the address of parameter i on the stack. The arithmetic stack now has two copies of the same value. The indexed fetch (ind) instruction on line 6 replaces the address placed on top of the stack with the corresponding value.

The operand of the ind instruction is an offset added to the address on top of the stack.

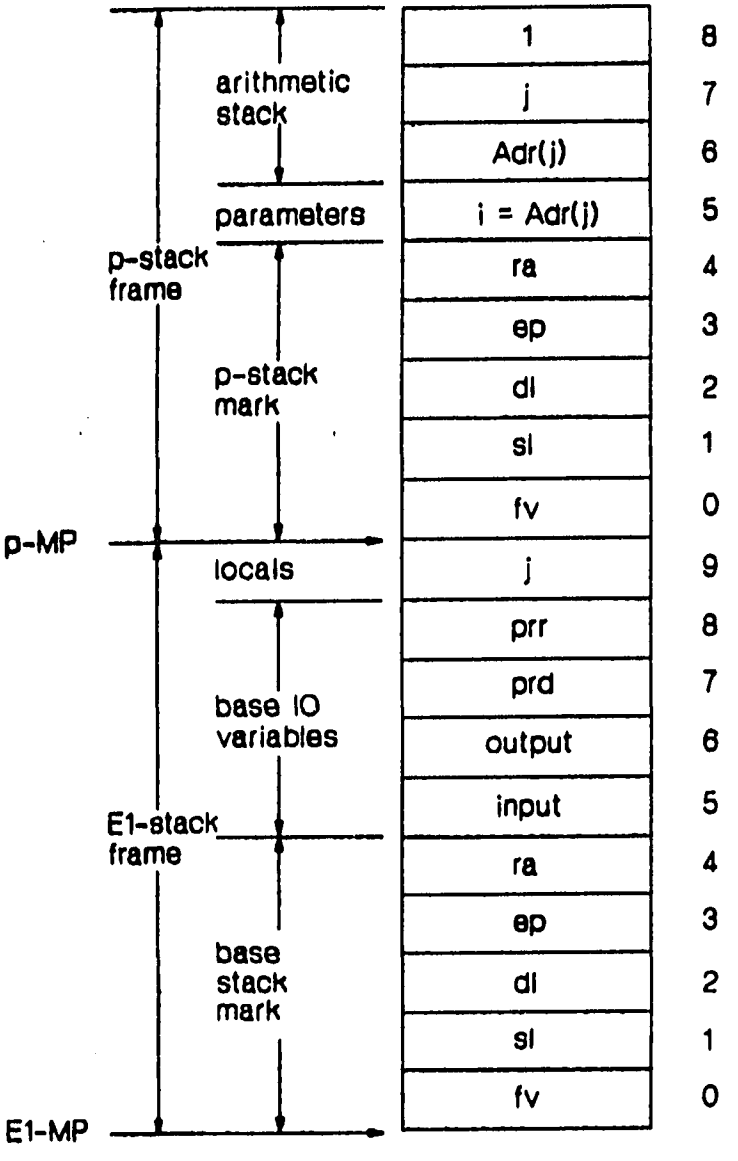


Figure 9. Stack for program E1

Line 7: The load constant (ldc) instruction places the constant in its operand on top of the stack. The arithmetic stack now contains three values and is at its maximum size. Figure 9 depicts the stack at this point. The stack pointer, SP, and extreme pointer, EP, are equal. The top of

stack contains the constant 1. The next to top of stack contains the value of variable j. The third and last value on the arithmetic stack contains the address of variable j.

Line 8: The add integer instruction (adi) replaces the top two values on the stack with their sum. The stack now contains the sum of i + 1 and the address of variable j.

Line 9: The store at base-level address (sto) consumes the top two values on the stack. The value on top of stack is assigned to the address contained in the value next to the top of stack.

Line 10: The return (ret) instruction causes control to be transferred to the calling procedure. The stack frame for the current procedure (p) is released. The program counter, PC, takes on the return address in the released stack mark. EP is restored and MP is assigned the value in the dynamic link of the released stack mark.

Line 21: Procedure p has just returned. The body of program E1 is finished. The return instruction transfers control to the prologue code which initialized the base stack mark.

Line 26: The stop instruction (stp) terminates execution.

A synopsis of P-code instructions is presented in Table 47. Complete definition for each P-code instruction is contained in the P-code interpreter written by K. Jensen, N. Wirth, and C. Jacobi [3]. An explanation of the P-code interpreter can be found in Pascal Implementation: Compiler and Assembler/Interpreter [3].

Table 47. P-code definitions

Mnemonic	Operation on stack		Parameters	Description
	Before	After		
abi	(i)	i		Absolute value of integer
abr	(r)	r		Absolute value of real
adi	(i,i)	i		Integer addition
adr	(r,r)	r		Real addition
chk C	No Change		PQ	Range check
chr	(i)	c		Converts integer to character
csp	special		Q	Call standard procedure
cup C	special		PQ	Call user procedure
dec C	(x)	x	Q	Decrement
dif	(s,s)	s		Set difference
dvi	(i,i)	i		Integer division
dvr	(r,r)	r		Real division
ent	special		PQ	Enter block
eof	(a)	b		End of file test
equ C	(x,x)	b	Q	Compare on equal
fjp	(b)			False jump
flo	(i,r)	r,r		Float next to top
flt	(i)	r		Float top of stack
geq C	(x,x)	b	Q	Compare on greater or equal
grt C	(x,x)	b	Q	Compare on greater than
inc C	(x)	x	Q	Increment
ind C	(a)	x	Q	Indexed fetch
inn	(i,s)	b		Set membership test
int	(s,s)	s		Set intersection
ior	(b,b)	b		Boolean inclusive OR
ixa	(a,i)	a	Q	Compute indexed address
lao		a	Q	Load base-level address
lca		a	Q	Load address of constant
lci		x	PQ	Load constant indirect - assembler generated
lda		a	PQ	Load address with level P
ldc C		x	Q	Load constant
ldo C		x	Q	Load contents of base-level address
leq C	(x,x)	b	Q	Compare on less or equal
les C	(x,x)	b	Q	Compare on less than
lod C		x	PQ	Load contents of address at level P

Table 47. Continued

Mnemonic	Operation on stack		Parameters	Description
	Before	After		
mod	(i,i)	i		Modulo
mov	(a,a)		Q	Move
mpi	(i,i)	i		Integer multiplication
mpr	(r,r)	r		Real multiplication
mst	special		P	Mark stack
neq C	(x,x)	b	Q	Compare on not equal
ngi	(i)	i		Integer sign inversion
ngr	(r)	r		Real sign inversion
not	(b)	b		Boolean not
odd	(i)	b		Test on odd
ord	(x)	i		Convert to integer
ret C	special			Return from block
sbi	(i,i)	i		Integer subtraction
sbr	(r,r)	r		Real subtraction
sgs	(i)	s		Generate singleton set
sqi	(i)	i		Square integer
sqr	(r)	r		Square real
sro C	(x)		Q	Store at base-level address
sto C	(a,x)			Store at base-level address
stp	No effect			Stop
str C	(x)		PQ	Store at level P
trc	(r)	i		Truncation, convert to integer
ujc	No effect			Error in case statement
ujp	No effect		Q	Unconditional jump
uni	(s,s)	s		Set union
xjp	(i)		Q	Indexed jump

5.3 The P-code Implementation

The purpose of this section is to discuss the principal components of the FPS-164 computer architecture and describe additional P-code instructions required for compilation and vector operations. An example is given to illustrate how P-code instructions are translated to Array Processor Assembly Language. Refer to the APAL64 Programmer's Reference Manual [30] and the APAL64 Programmer's Guide [35] for detailed information on the FPS-164 Scientific Computer.

Functional units and data paths of the FPS-164 are shown in Figure 10. The FPS-164 has two memories, three functional units, and three register files, all of which can be active in a single instruction cycle.

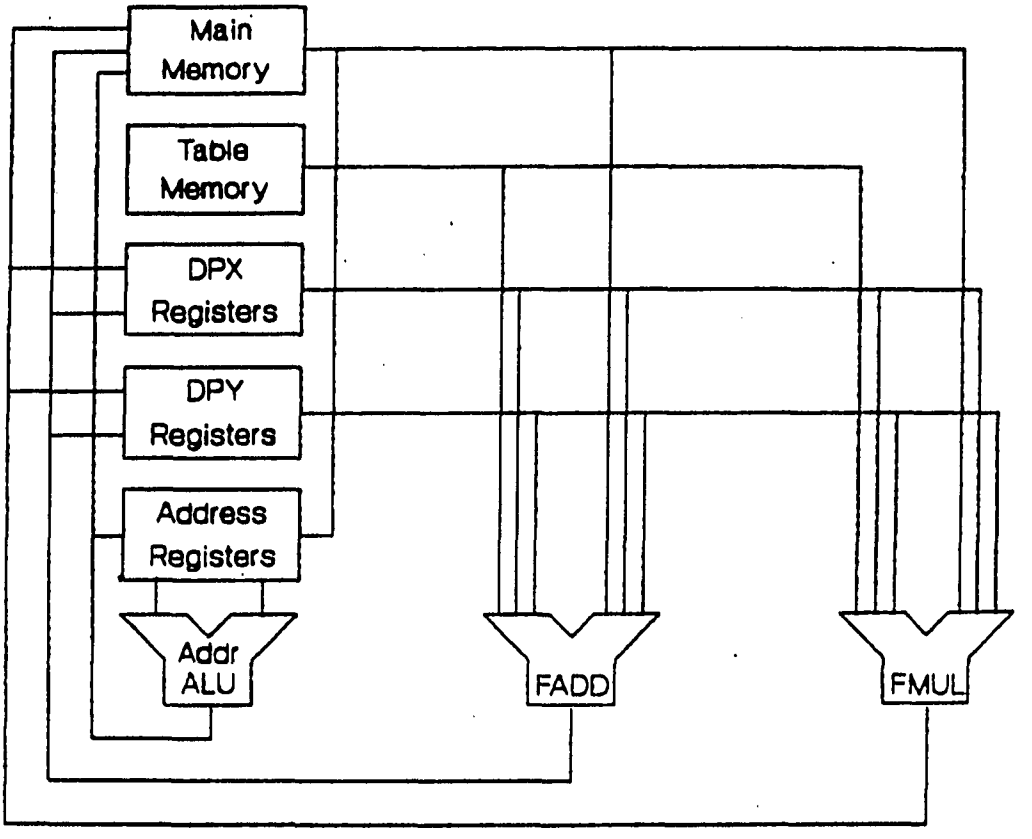


Figure 10. Major components of the FPS-164 architecture

Main Memory has a three-cycle latency and Table Memory has a two-cycle latency. If the Main Memory address register is set, then three instruction cycles later the corresponding data will be available and, likewise, if the Table Memory address register is altered, corresponding data will be available two cycles later. All of Main Memory can be read and altered whereas the first 5,078 words of Table Memory contain constants and cannot be changed. Additional read-write Table Memory can be purchased from Floating Point

Systems. The smallest addressable unit of memory is a 64-bit word. Memory addresses occupy 24 bits.

The FPS-164 is designed to execute only one job in an interval. FPS Main Memory contains exactly two executable images: Single User Monitor and the user's program.

Memory address registers for both Main Memory and Table Memory are connected to the output of the Address arithmetic and logic unit (ALU). Data from both memories can be assigned to any address register, data pad register, the floating point adder, FADD, or the floating point multiplier, FMUL.

Register files shown in Figure 10 are called data pads address registers. Data pad registers are divided into two groups of 32 registers, DPX and DPY. An element of both groups can be read and altered in an instruction cycle. Reading occurs at the beginning of the cycle and alteration (writing) occurs at the end of the cycle. Data pad registers are composed of 64-bits and intended to be used in floating point operations. Fixed point and logical operations can also be performed on 64-bit data.

A local bus interconnects DPX and DPY permitting data to be copied from one data pad to the other directly. Input and output latches of the floating point units, FADD and FMUL, are also directly connected to the data pad registers.

Sixty-four address registers are directly connected to the Address ALU. Each address register is composed of 32 bits. Address registers are primarily designed to store addresses (24 bits), however, any fixed point data can be retained in an address register.

The address ALU, labeled Addr ALU in Figure 10, can perform arithmetic and logic functions on 32-bit data.

The floating point adder (FADD) can be used to add floating point data, add fixed point values, and perform logical operations on 64-bit data. The floating point adder

has a two-cycle latency. Results from the floating point adder are available on the instruction cycle following the second activation of the adder. The floating point adder does not need to be executed on sequential instruction cycles. Partially formed results are retained indefinitely by the the adder.

The floating point multiplier (FMUL) performs fixed and floating point multiplications. Results from the floating point multiplier are available on the instruction cycle following the third instruction in which the multiplier executes. The floating point multiplier does not need to be executed in every instruction cycle. Partially formed results are retained indefinitely.

The reader is invited to focus on the problems of translating P-codes to Array Processor Assembly Language now that an overview of the FPS-164 architecture has been presented. Main Memory can be partitioned into instructions and data. P-code instructions are translated into equivalent APAL statements and placed in the instruction-partition of Main Memory. The stack, heap, and selected constants are allocated in the data-partition of Main Memory as shown in Figure 11.

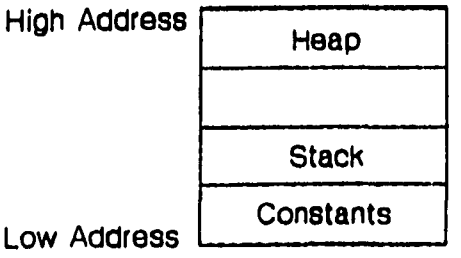


Figure 11. Vector Pascal data organization on the FPS-164

Constant directives, provided by the the FPS Assembler, APAL64, make it convenient to allocate constants below the stack (see Figure 11). Constants are allocated above the

heap in the P-machine (see Figure 6). The heap and stack retain their original relationship.

The FPS-164 has a subroutine stack which functions like a stack mark except the subroutine stack contains only the return address. The return address is placed on the subroutine stack by APAL subroutine-jump instructions (jsr). Dynamic and static links are not put on the subroutine stack and these are needed to restore the mark pointer register and access non-local data. Likewise, no provision for either the return value or the extreme pointer is made.

Figure 12 shows the format of a stack frame on the FPS-164. Starting at the top of the stack frame in Figure 12 the reader can see that the arithmetic stack does not appear. Recall the extreme pointer marks the extent of the arithmetic stack. The extreme pointer and the arithmetic stack are eliminated by using address registers and data pad registers in place of the arithmetic stack. Note the old extreme pointer does not appear in the stack mark. In place of the old extreme pointer is the stack size (ss) which has no function other than to record the number of words allocated to the stack mark (5), parameters and local variables.

Originally, Vector Pascal contained a basic type of complex. Type complex was a template for a complex variable composed of two real numbers, one for the real portion of the complex number and the other for the imaginary portion. To support the complex type, two slots are allocated in the stack mark for a complex variable (see Figure 12). The slots are labeled rv1 and rv2. The slot denoted rv2 was intended for the imaginary portion of a complex variable. The slot identified as rv1 is used in the same way as the function value (fv) slot (see Figure 8) is used on a P-machine. Type complex was never implemented.

Static links permit non-local variables to be accessed from the local environment. In order to fetch a non-local variable, the static link chain must be followed to the stack frame in which the non-local variable resides. The P-parameter on selected P-code instructions indicates how many links to traverse to get to the referenced stack frame.

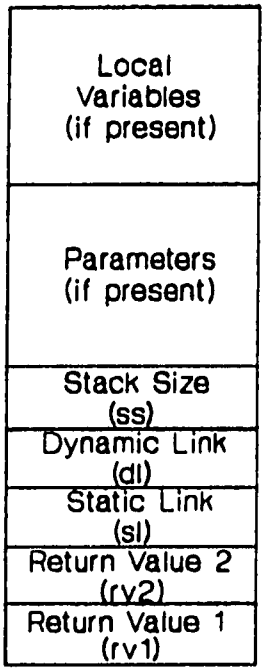


Figure 12. Vector Pascal stack frame

It is desirable to avoid the traversal process since each link would require reading Main Memory. Three instruction cycles are needed each time Main Memory is read to get to the next link.

Instead of traversing the static link list a static link display is used. Eight address registers are allocated to a static link display. The P-parameter designates which

of the eight registers points to the base of the desired stack frame. Gries [25] discusses performance benefits gained by using a display to fetch non-local data. Eight registers permit procedure nesting up to eight levels deep.

The static link (sl) slot in Figure 12 is used to identify which of the mark pointer registers to restore when a procedure returns to its caller.

The dynamic link (dl) pointer in Figure 12 points to the base of the previous stack frame as it does for the P-machine (see Figure 8).

All parameter and variable addresses are relative to the base of the stack frame in which they are allocated. Altering the stack mark size would shift addresses by the amount the stack mark was increased or decreased. The compiler assigns addresses and it was convenient to make the stack mark size used by Vector Pascal on the FPS-164 identical to the stack mark size on the P-machine.

Important highlights of Vector Pascal data organization on the FPS-164 include the following. Constants are allocated below the stack. Address and data pad registers are used to replace the arithmetic stack on the P-machine. As a result of replacing the arithmetic stack with registers the extreme pointer is not needed and was, therefore, discarded. The function provided by static links is replaced by a static link display and the static link slot in the stack mark is used to maintain the display.

The reader is now invited to consider additional P-codes required to support the model of data organization presented above. Recall the P-parameter on selected P-code instructions indicates the number of static links to traverse to find the stack frame of the reference variable. The P-parameter is computed by subtracting the level of the desired stack frame from the current level. Hence the P-parameter is relative to the current level. In contrast,

the static link display represents actual lexical levels. The P-parameter is used by the P-code to APAL Assembler to determine which static link display mark pointer register to use. The P-machine P-code instruction set does not indicate the current lexical level. To solve this problem, the lexical (lex) P-code was invented. The lex P-code instruction identifies the current lexical level (see Table 48). The lexical level established by the lex P-code instruction remains in effect until the next lexical instruction is encountered or the program stops. The lex P-code is emitted at the start.

The P4 version of the Pascal compiler does not include standard function round. Vector Pascal supports standard function round. As a result, the round (rnd) P-code instruction was added. The round P-code converts the real number on top of stack to an integer by rounding. Table 48 summarizes the functions of the P-code instructions lex and rnd.

The reader is now invited to consider additions to the P-code instruction set to facilitate vector operations. This discussion also identifies the key concepts necessary to translate vector P-code instructions to APAL.

The problem is to devise operands which descibe vectors and operands which perform vector functions. The vector operators and operands should fit within the P-code paradigm. The operand must describe a vector originating as a slice of a matrix or a vector that is an array having a single index.

Vector Pascal allocates arrays in sequential, row-major order. A key characteristic of this arrangement is that sequential elements in a dimension of an array are a fixed distance apart. The distance separating elements in a dimension is called the stride or step. For example, consider an array having ten rows and five columns. If the

array is allocated in row-major order then elements in a row are next to each other and neighboring rows are five elements apart. The row-stride is the distance to the next element in the row and in the example above the row-stride is equal to one element. The column-stride is the distance to the next element in the column and in the example above the column-stride is equal to five elements. Knuth [36] describes algorithms for locating elements in sequentially allocated arrays from an arbitrary set of indexes.

Table 48. P-code additions

Mnemonic	Operation on stack Before	After	Param-eters	Description
adv C	(v,v)	v		Vector addition
lex	No Change		Q	Identify static lexical level
ldv C	(a,i,i,i,i)	v		Load vector descriptor
msv	(n,v)	v		Multiply scalar vector
mvs	(v,s)	v		Multiply vector scalar
mvv	(v,v)	n		Multiply vector vector
ndx	(a,i)	a	PQ	Compute indexed address with stride
rlu	special		Q	Roll up Q elements on stack
rnd	(r)	i		Round, convert to integer
stv C	(v,v)			Store vector

Since elements of rows and columns are allocated at regular intervals, a vector can be described with the information in Table 49. The base address is the address of the beginning of a row or column in which the vector occurs. The zero-bias is the low bound in the array declaration for that dimension. The zero-bias is subtracted from an index to compute the offset, in steps, to the referenced element. The high index and low index values identify the extent of the vector. The stride is the distance between elements. The element size is known to be one since both real numbers and integers occupy one word. Integers and real numbers are

the only valid base types of vectors. Summarizing, a vector descriptor is an address followed by four integers which describes a vector having elements spaced at regular intervals.

Table 49. Vector additions to primitive types

Mnemonic	Primitive Type	
n	numeric, real or integer	
v	vector descriptor consisting of the following:	
a	base address of vector,	tos
i	stride	tos-1
i	zero-bias	tos-2
i	high index	tos-3
i	low index	tos-4

P-code instructions which perform vector operations can be discussed now that an understanding of a vector descriptor has been established. Table 49 lists P-code instructions which perform vector operations.

The add-vector instruction (adv) adds two vectors together and places the vector descriptor of their sum on top of the stack. Corresponding elements in the addend and augend vectors are added and their sums are placed in the resulting vector.

The load-vector instruction (ldv) encapsulates the top five elements on the stack into a vector descriptor. No actual work is performed.

The multiply-scalar-vector instruction (msv) has two operands. The top of stack contains a vector descriptor and next to the top is a real or integer. The C-parameter (see Table 48) determines the type of the vector and the scalar. The msv instruction multiplies every element in operand vector by the scalar operand. The resulting vector descriptor identifies the scaled vector.

The multiply-vector-scalar instruction (mvs) performs the same function as the msv instruction. The only

difference is that the vector descriptor and scalar operands are reversed.

The multiply-vector-vector instruction (mvv) performs the inner product. The scalar result placed on top of the stack.

The store-vector instruction places the vector described by the vector descriptor next to the top of the stack in the location described by the vector descriptor on top of the stack.

The roll-up instruction (rlu) makes a temporary circular list out of the top Q elements on the stack. The list is then shifted one element such that the top element is now the bottom element and the next to top element is the top element. This instruction is used in the construction of a vector descriptor.

One final difference between the P-machine version of P-code instructions and the Vector Pascal version is addresses. All addresses in the Vector Pascal version of the P-code instruction set are character or byte addresses. Results of Vector Pascal programs are transformed into characters by Vector Pascal standard procedures to read and write data. Eight characters are packed into one FPS-164 word.

Fundamentally, vector operations are identified in Vector Pascal syntax. When the Vector Pascal compiler recognizes a vector operation, corresponding P-code instructions are generated. The P-code to APAL Assembler is similar to the Vector Pascal compiler. The Assembler recognizes one or more P-code instructions that use vector operations and emits a call to the corresponding subroutine.

P-code instructions which do not perform vector operations are immediately translated into equivalent APAL statements.

In contrast, P-code instruction which perform vector operations are not translated directly into APAL statements. Instead vector P-code instructions are stacked until the result of a vector operation is a scalar or until a store vector (stv) P-code is encountered. The foregoing algorithm solves the problem of vector temporaries and has the benefit of improving performance over other solutions.

Consider the problem of vector temporaries. The add vector (adv) and multiply-scalar-vector (msv and mvs) P-code instructions produce a vector descriptor. The storage for the resulting vector associated with the vector descriptor is not defined. The resulting vector is temporary: it is part of the vector expression on the right hand side of a Vector Pascal statement. Eventually the vector temporary will be discarded.

The reader is invited to examine some possibilities for allocating storage to vector temporaries. Storage for the vector temporary could be allocated from the arithmetic stack. This solution must be rejected because vectors can be arbitrarily large and could exceed the number of registers available for the arithmetic stack.

Vector operations are performed in a repetitive loop. Elements of the operands are processed individually. The key to performance is to execute the loop management and address arithmetic used to fetch operands in parallel with the floating point computations of the vector operation. For most vector operations more instructions are executed in loop management, address arithmetic, and accessing memory than in actual floating point operations. It is desirable, therefore to perform many floating point operations on the elements of the operand vectors while they are in registers before returning the results to Main Memory.

Another solution is to allocate space for the vector temporary from the heap. However, the cost incurred to manage vector temporaries makes this solution unattractive.

The solution to vector temporaries is, then, to never allocate a vector temporary. The benefits are: storage requirements are reduced over other solutions and, for complex vector expressions, performance is increased over other solutions. One disadvantage is increased complexity in the P-code to APAL Assembler. Vector P-code instructions must be recorded so they can be processed when a store vector instruction is executed.

5.4 A Vector Pascal Example

The purpose of this section is to illustrate how a Vector Pascal program is translated into Array Processor Assembler Language. A simple Vector Pascal program is explained along with its translations into P-code instructions and Array Processor Assembler Language Statements.

The example program shown in Table 50 contains the statement most often executed in LINPACK (lines 16 and 17). Table 51 contains the P-code instructions which correspond to Vector Pascal statements in Table 50. Vector Pascal statements are printed as comments just before corresponding P-code instructions in Table 51. The APAL version of Vector Pascal program E2 is listed in Table 52. Comments produced by the P-code to APAL Assembler in Table 52 have been truncated to fit within the margins allowed for dissertations.

Declarations and other statements in Vector Pascal program E2 (see Table 50) support the Vector Pascal statement

```
A[r,c+1..rank]:= A[r,c+1..rank] +
                 A[r,c]*A[pivotrow,c+1..rank];
```

which appears on lines 16 and 17. The statement performs the same function as DAXPY (see Table 31). The statement is executed $O(n^2)$ times by LINPACK and $O(n^3)$ floating point operations are performed during execution. The vector expression on the right side of the assignment token (:=) is the sum of two row slices. The second row-slice is scaled by the multiplier, A[r,c]. The vector expression is assigned to the row-slice on the left of the assignment token.

Table 50. Vector Pascal program E2

```
Line Pascal Statement
1    program E2;
2      const
3        rank=10;
4      type
5        idxrng  = 1..rank;
6        rvector = array[idxrng] of real;
7        matrix  = array[idxrng] of rvector;
8      var
9        r,c,pivotrow: idxrng;
10       A: matrix;
11       job: integer;
12   begin{E2}
13     pivotrow := 3;
14     c := 3;
15     for r:=pivotrow+1 to rank do
16       A[r,c+1..rank]:= A[r,c+1..rank] +
17                        A[r,c]*A[pivotrow,c+1..rank];
18   end{E2}.
```

Table 51. P-code form of program E2

```
Line P-code Instructions
1    c
2    c program E2;
3    c   const
4    c     rank=10;
5    c   type
6    c     idxrng  = 1..rank;
7    c     rvector = array[idxrng] of real;
8    c     matrix  = array[idxrng] of rvector;
```

Table 51. Continued

```
Line P-code Instructions
9     c   var
10    c     r,c,pivotrow: idxrng;
11    c     A: matrix;
12    c     job: integer;
13    c begin{E2}
14    1   3
15     lex            1
16     ent   1   1    4
17     ent   2   1    5
18    c   pivotrow := 3;
19     ldci                3
20     chki  1       10
21     sroi          72
22    c   c := 3;
23     ldci                3
24    i   10
25     chki  1       10
26     sroi          80
27     ldoi          72
28     ldci                1
29     adi
30     sroi          88
31    c   for r:=pivotrow+1 to rank do
32     ldci              10
33     stri  0     904
34    l   6
35     ldoi           88
36     lodi  0     904
37    i   20
38     leqi
39     fjp        1   7
40     lao           96
41     ldoi          88
42     chki  1       10
43     deci           1
44     ixa           80
45     ldoi          80
46     ldci                1
47     adi
48    i   30
49     ldci             10
50     ldci           1
51     ldci           8
52     rlu            5
53     ldvr
54     lao           96
55     ldoi          88
```

Table 51. Continued

```
Line P-code Instructions
56    chki  1       10
57    deci           1
58    ixa           80
59   i   40
60    ldoi          80
61    ldci                1
62    adi
63    ldci               10
64    ldci           1
65    ldci           8
66    rlu            5
67    ldvr
68    lao           96
69    ldoi          88
70   i   50
71    chki  1       10
72    deci           1
73    ixa           80
74    ldoi          80
75    chki  1       10
76    deci           1
77    ixa            8
78    indr           0
79    lao           96
80    ldoi          72
81   i   60
82    chki  1       10
83    deci           1
84    ixa           80
85    ldoi          80
86    ldci                1
87    adi
88    ldci               10
89    ldci           1
90    ldci           8
91    rlu            5
92   c     A[r,c+1..rank] := A[r,c+1..rank] +
93   c                       A[r,c]*A[pivotrow,c+1..rank];
94   i   70
95    ldvr
96    msvr
97    advr
98    stvr
99    ldoi          88
100   inci           1
101   sroi          88
102   ujp       1   6
```

Table 51. Continued

```
Line P-code Instructions
103  l   7
104  c end{example2}.
105   retp
106  l   4=            912
107  l   5=             71
108  q
109  i    0
110   mst            0
111   cupp  0   1    3
112   stp
113  q
```

Translation of P-code instructions which implement Vector Pascal program E2 are discussed in the following paragraphs. P-code instructions are generated for executable statements starting on line 12, Table 50. The prologue to the body of program E2 is on lines 14 through 17. Lines 18 through 26 contain P-code instructions which initialize variables pivotrow and c. P-code instructions on lines 27 through 39 initialize the loop variable, r, and test variable r against the loop limit (constant rank). The Vector Pascal statement on line 16 and 17 in Table 50 is translated to P-code instructions on lines 40 through 98 in Table 51. Each line is discussed. Lines 98 through 101 in Table 51 contain P-code instructions which increment the loop variable, r, and transfer control to the loop test beginning on line 34. The remaining lines, 102 through 112, constitute the program prologue. The following paragraphs discuss in detail the P-codes which implement the vector assignment statement on lines 16 and 17 of Table 50.

Lines 40 through 53: The vector descriptor for the row-slice on the left side of the assignment statement (lines 16 and 17, Table 50) is created.

Line 40: The base address of matrix A is pushed on top of the stack.

Lines 41 through 43: The bias on the row index, r, is removed.

Line 44: The base address of the row slice is computed by adding the row offset to the base address of matrix A. The row offset is computed by multiplying the operand of the ixa instruction (80) by the unbiased row index.

Lines 45 through 47: The lowest index of the row-slice is computed. The column index, c, is fetched on line 45. Load constant (ldc) and add integer (adi) instructions on lines 46 and 47 increment the lowest index of the row-slice.

Line 49: The highest index of the row-slice is placed on the stack.

Line 50: The zero-bias is put on the stack.

Line 51: The stride, measured in bytes, is put on the stack.

Line 52: The vector descriptor is now on the stack. However, it is in the wrong order. The roll-up instruction shifts the stack so that the base address of the vector is put on top of the stack.

Line 53: The load vector (real) encapsulates the vector descriptor.

Lines 54 through 67: The vector descriptor for the first vector on the right-hand side of the assignment statement (lines 16 and 17 of Table 50) is created. These lines are identical to lines 40 through 53.

Lines 68 through 78: The multiplier, A[r,c], in the vector expression (lines 16 and 17, Table 50) is put on the stack.

Line 68: The base address of matrix A is fetched.

Lines 69, 71 and 72: The bias on the row index, r, is removed.

Line 73: The base address of row r in matrix A is computed. First the offset to row r is computed by

multiplying the row size by the unbiased row index. Then the product is added to the base address of matrix A.

Lines 74 through 76: The column index, c, is transformed into an offset to column c in row r.

Line 77: The address of A[r,c] is computed by adding the column offset to the base address of the row in the same way as the base address of the row was computed on lines 44, 58, and 73.

Line 78: The address of the multiplier, A[r,c], is replaced by its value.

Lines 79 through 94: The vector descriptor of the pivot row-slice is created. This section of P-code instructions is identical to P-code instructions on lines 40 through 47 except line 80 which fetches variable pivotrow instead of row r.

Line 95: The pivot row row-slice is scaled by the multiplier, A[r,c].

Line 96: The row-slice corresponding to row r and the scaled pivot row row-slice are added.

Line 97: The sum computed on the previous line is stored in the row r row-slice.

The reader is now invited to consider the Array Processor Assembler Language form of program E2 in Table 52. P-code instructions in the comment fields are translated to APAL statements which follow. A synopsis of the program is presented in the following paragraphs.

Lines 1 through 52: The standard prologue for Vector Pascal programs is shown on lines 1 through 50. The prologue contains directives which allow the program to be executed under the control of SJE and SUM. External declarations for procedures and functions in the Vector Pascal run-time library are listed on lines 4 through 27. The stack mark and static link display are defined on lines 34 through 50.

Table 52. APAL form of program E2

```
Line APAL Statements                                          Comments
1               $TITLE $MAIN$
2               $insert'=PS.KEY'
3               $insert'=SYS.KEY'
4               $insert'=TM.KEY'
5               $ext wln
6               $ext wrc
7               $ext wri
8               $ext wrr
9               $ext wrs
10              $ext rln
11              $ext rdc
12              $ext rdi
13              $ext rdr
14              $ext ma
15              $ext vs
16              $ext ip
17              $ext swap
18              $ext AOP$I_IDIV
19              $ext AOP$I_RDIV
20              $ext FIF$I_MOD
21              $ext FIF$I_SIN
22              $ext FIF$I_COS
23              $ext FIF$I_ATAN
24              $ext FIF$I_ALOG
25              $ext FIF$I_EXP
26              $ext FIF$I_SQRT
27              $ext RANDOM
28              $ext CLOCK
29              $ENTRY .MAIN.
30              $ENTRY $MAIN$
31              $HSUBR $MAIN$
32              $PSECT .CODE.,PS ,CAT
33
34                                                          "stack mar
35    rv1       $equ      0                                 "return va
36    rv2       $equ      1                                 "return va
37    sl        $equ      2                                 "static li
38    dl        $equ      3                                 "display l
39    ss        $equ      4                                 "stack siz
40                                                          "static li
41    mp0       $equ      sp(32)                             "mark poi
42    mp1       $equ      sp(33)                             "mark poi
43    mp2       $equ      sp(34)                             "mark poi
44    mp3       $equ      sp(35)                             "mark poi
45    mp4       $equ      sp(36)                             "mark poi
46    mp5       $equ      sp(37)                             "mark poi
```

Table 52. Continued

```
Line  APAL Statements                                  Comments
47    mp6       $equ       sp(38)                     "mark poi
48    mp7       $equ       sp(39)                     "mark poi
49    nmp       $equ       sp(40)                     "new mark
50
51    start:    $LOC .+11
52              jmp begin
53    13:                                             "1   3
54                                                    " lex
55              mov nmp,mp0                           "
56                                                    " ent   1
57                                                    " ent   2
58                                                    " ldci
59              dpx<3                                 "
60              dpx(-8)<dpx                           "
61                                                    " chki  1
62                                                    " sroi
63              ldspi sp(41);db=72                    "
64              addrr mp0,sp(41)                      "
65              movr sp(41),sp(41);setma              "
66              statma;mi<dpx(-8)                     "
67                                                    " ldci
68              dpx<3                                 "
69              dpx(-8)<dpx                           "
70                                                    " chki  1
71                                                    " sroi
72              ldspi sp(41);db=80                    "
73              addrr mp0,sp(41)                      "
74              movr sp(41),sp(41);setma              "
75              statma;mi<dpx(-8)                     "
76                                                    " ldoi
77              ldspi sp(41);db=72                    "
78              add mp0,sp(41)                        "
79              movrr sp(41),sp(41)                   "
80              movr sp(41),sp(41);setma              "
81              nop                                   "
82              nop                                   "
83              dpx(-8)<md                            "
84                                                    " ldci
85              dpx<1                                 "
86              dpy(-8)<dpx                           "
87                                                    " adi
88              fiadd dpy(-8),dpx(-8)                 "
89              fapush                                "
90              dpy(-8)<fa                            "
91                                                    " sroi
92              ldspi sp(41);db=88                    "
```

Table 52. Continued

Line	APAL Statements	Comments
93	addrr mp0,sp(41)	"
94	movr sp(41),sp(41);setma	"
95	statma;mi<dpy(-8)	"
96		"
97		" ldci
98	dpx<10	"
99	dpx(-8)<dpx	"
100		" stri 0
101	ldspi sp(41);db=904	"
102	addrr mp0,sp(41)	"
103	movr sp(41),sp(41);setma	"
104	statma;mi<dpx(-8)	"
105	16:	"1 6
106		" ldoi
107	ldspi sp(41);db=88	"
108	add mp0,sp(41)	"
109	movrr sp(41),sp(41)	"
110	movr sp(41),sp(41);setma	"
111	nop	"
112	nop	"
113	dpx(-8)<md	"
114		" lodi 0
115	ldspi sp(41);db=904	"
116	add mp0,sp(41)	"
117	movrr sp(41),sp(41)	"
118	movr sp(41),sp(41);setma	"
119	nop	"
120	nop	"
121	dpy(-8)<md	"
122		" leqi
123	ficmp dpx(-8),dpy(-8)	"
124	fapush	"
125	nop	"
126	bfle t0	"
127	f0: movi 0,sp(41)	"
128	br e0	"
129	t0: movi 1,sp(41)	"
130	e0:	
131		" fjp
132	mov sp(41),sp(41)	"
133	jmpeq 17	"
134		" lao
135	ldspi sp(42);db=96	"
136	add mp0,sp(42)	"
137		" ldoi
138	ldspi sp(43);db=88	"

Table 52. Continued

Line	APAL Statements	Comments
139	add mp0,sp(43)	"
140	movrr sp(43),sp(43)	"
141	movr sp(43),sp(43);setma	"
142	nop	"
143	nop	"
144	dpx(-8)<md	"
145		" chki 1
146		" deci
147	dpy<1	"
148	fisub dpx(-8),dpy	"
149	fapush	"
150	dpx(-8)<fa	"
151		" ixa
152	dpy<dpx(-8)	"
153	dpx<80	"
154	fimul dpx,dpy	"
155	fmpush	"
156	fmpush	"
157	dpy<fm	"
158	ldspi sp(44);db=dpy	"
159	add sp(44),sp(42)	"
160		" ldoi
161	ldspi sp(43);db=80	"
162	add mp0,sp(43)	"
163	movrr sp(43),sp(43)	"
164	movr sp(43),sp(43);setma	"
165	nop	"
166	nop	"
167	dpx(-8)<md	"
168		" ldci
169	dpx<1	"
170	dpy(-8)<dpx	"
171		" adi
172	fiadd dpy(-8),dpx(-8)	"
173	fapush	"
174	dpy(-8)<fa	"
175		" ldci
176	dpx<10	"
177	dpx(-8)<dpx	"
178		" ldci
179	dpx<1	"
180	dpx(-7)<dpx	"
181		" ldci
182	dpx<8	"
183	dpy(-7)<dpx	"
184		" rlu

Table 52. Continued

Line	APAL Statements	Comments
185		" ldvr
186		" lao
187	ldspi sp(43);db=96	"
188	add mp0,sp(43)	"
189		" ldoi
190	ldspi sp(44);db=88	"
191	add mp0,sp(44)	"
192	movrr sp(44),sp(44)	"
193	movr sp(44),sp(44);setma	"
194	nop	"
195	nop	"
196	dpx(-6)<md	"
197		" chki 1
198		" deci
199	dpy<1	"
200	fisub dpx(-6),dpy	"
201	fapush	"
202	dpx(-6)<fa	"
203		" ixa
204	dpy<dpx(-6)	"
205	dpx<80	"
206	fimul dpx,dpy	"
207	fmpush	"
208	fmpush	"
209	dpy<fm	"
210	ldspi sp(45);db=dpy	"
211	add sp(45),sp(43)	"
212		" ldoi
213	ldspi sp(44);db=80	"
214	add mp0,sp(44)	"
215	movrr sp(44),sp(44)	"
216	movr sp(44),sp(44);setma	"
217	nop	"
218	nop	"
219	dpx(-6)<md	"
220		" ldci
221	dpx<1	"
222	dpy(-6)<dpx	"
223		" adi
224	fiadd dpy(-6),dpx(-6)	"add two o
225	fapush	"
226	dpy(-6)<fa	"
227		" ldci
228	dpx<10	"
229	dpx(-6)<dpx	"
230		" ldci

Table 52. Continued

Line	APAL Statements	Comments
231	dpx<1	"
232	dpx(-5)<dpx	"
233		" ldci
234	dpx<8	"
235	dpy(-5)<dpx	"
236		" rlu
237		" ldvr
238		" lao
239	ldspi sp(44);db=96	"
240	add mp0,sp(44)	"
241		" ldoi
242	ldspi sp(45);db=88	"
243	add mp0,sp(45)	"
244	movrr sp(45),sp(45)	"
245	movr sp(45),sp(45);setma	"
246	nop	"
247	nop	"
248	dpx(4)<md	"
249		" chki 1
250		" deci
251	dpy<1	"
252	fisub dpx(4),dpy	"
253	fapush	"
254	dpx(4)<fa	"
255		" ixa
256	dpy<dpx(4)	"
257	dpx<80	"
258	fimul dpx,dpy	"
259	fmpush	"
260	fmpush	"
261	dpy<fm	"
262	ldspi sp(46);db=dpy	"
263	add sp(46),sp(44)	"
264		" ldoi
265	ldspi sp(45);db=80	"
266	add mp0,sp(45)	"
267	movrr sp(45),sp(45)	"
268	movr sp(45),sp(45);setma	"
269	nop	"
270	nop	"
271	dpx(4)<md	"
272		" chki 1
273		" deci
274	dpy<1	"
275	fisub dpx(4),dpy	"
276	fapush	"

Table 52. Continued

Line	APAL Statements	Comments
277	dpx(4)<fa	"
278		" ixa
279	dpy<dpx(4)	"
280	dpx<8	"
281	fimul dpx,dpy	"
282	fmpush	"
283	fmpush	"
284	dpy<fm	"
285	ldspi sp(46);db=dpy	"
286	add sp(46),sp(44)	"
287		" indr
288	addi 0,sp(44)	"
289	movrr sp(44),sp(44)	"
290	movr sp(44),sp(44);setma	"
291	nop	"
292	nop	"
293	dpx(4)<md	"
294		" lao
295	ldspi sp(44);db=96	"
296	add mp0,sp(44)	"
297		" ldoi
298	ldspi sp(45);db=72	"
299	add mp0,sp(45)	"
300	movrr sp(45),sp(45)	"
301	movr sp(45),sp(45);setma	"
302	nop	"
303	nop	"
304	dpy(4)<md	"
305		" chki 1
306		" deci
307	dpx<1	"
308	fisub dpy(4),dpx	"
309	fapush	"
310	dpy(4)<fa	"
311		" ixa
312	dpx<80	"
313	fimul dpx,dpy(4)	"
314	fmpush	"
315	fmpush	"
316	dpy(4)<fm	"
317	ldspi sp(46);db=dpy(4)	"
318	add sp(46),sp(44)	"
319		" ldoi
320	ldspi sp(45);db=80	"
321	add mp0,sp(45)	"
322	movrr sp(45),sp(45)	"

Table 52. Continued

Line	APAL Statements	Comments
323	movr sp(45),sp(45);setma	"
324	nop	"
325	nop	"
326	dpy(4)<md	"
327		" ldci
328	dpx<1	"
329	dpx(5)<dpx	"
330		" adi
331	fiadd dpx(5),dpy(4)	"
332	fapush	"
333	dpx(5)<fa	"
334		" ldci
335	dpx<10	"
336	dpy(4)<dpx	"
337		" ldci
338	dpx<1	"
339	dpy(5)<dpx	"
340		" ldci
341	dpx<8	"
342	dpx(6)<dpx	"
343		" rlu
344		"
345		" ldvr
346		" msvr
347		" advr
348		" stvr
349	mov sp(44),sp(1)	"
350	dpy(-4)<dpy(5)	"
351	dpy(-1)<dpy(4)	"
352	dpx(-1)<dpx(5)	"
353	dpx(-4)<dpx(6)	"
354	dpx(0)<dpx(4)	"
355	mov sp(43),sp(2)	"
356	dpy(-3)<dpx(-5)	"
357	dpy(0)<dpx(-6)	"
358	dpx(1)<dpy(-6)	"
359	dpx(-3)<dpy(-5)	"
360	mov sp(42),sp(0)	"
361	dpy(-2)<dpx(-7)	"
362	dpy(1)<dpx(-8)	"
363	dpx(2)<dpy(-8)	"
364	dpx(-2)<dpy(-7)	"
365	jsr ma	"Multiply
366		" ldoi
367	ldspi sp(42);db=88	"
368	add mp0,sp(42)	"

Table 52. Continued

Line	APAL Statements	Comments
369	movrr sp(42),sp(42)	"
370	movr sp(42),sp(42);setma	"
371	nop	"
372	nop	"
373	dpx(-8)<md	"
374		" inci
375	dpy<1	"
376	fiadd dpx(-8),dpy	"
377	fapush	"
378	dpx(-8)<fa	"
379		" sroi
380	ldspi sp(42);db=88	"
381	addrr mp0,sp(42)	"
382	movr sp(42),sp(42);setma	"
383	statma;mi<dpx(-8)	"
384		" ujp
385	jmp 16	"
386	17:	"1 7
387		"
388		" retp
389	movrr mp0,sp(42)	"
390	movr sp(42),sp(42)	"
391	addi# d1,sp(42);setma	"
392	addi# s1,sp(42);setma	"
393	nop	"
394	ldtma;db=md	"
395	ldspi sp(1);db=md	"
396	jmpt rstrmp	"
397		"1 4=
398		"1 5=
399	begin:	
400		" mst
401	ldspi nmp;db=stack	"
402	addi# s1,nmp;setma	"
403	statma;mi<zero	"initial s
404	incma;mi<zero	"initial d
405	incma;mi<zero	"initial s
406	incma;movi# 5,sp(0);mi<spfn	"input fil
407	incma;movi# 6,sp(0);mi<spfn	"output fi
408	movl nmp,nmp	"
409	movl nmp,nmp	"
410	movl nmp,nmp	"
411		" cupp 0
412	jsr 13	"
413		" stp
414	return	"return to

Table 52. Continued

```
Line APAL Statements                                        Comments
415  rstrmp:  mov sp(1),mp0                                 "
416           return                                        "return to
417           mov sp(1),mp1                                 "
418           return                                        "return to
419           mov sp(1),mp2                                 "
420           return                                        "return to
421           mov sp(1),mp3                                 "
422           return                                        "return to
423           mov sp(1),mp4                                 "
424           return                                        "return to
425           mov sp(1),mp5                                 "
426           return                                        "return to
427           mov sp(1),mp6                                 "
428           return                                        "return to
429           mov sp(1),mp7                                 "
430           return                                        "return to
431                     $LOC start+9
432                     $ASCII '$MAIN$'
433                     $HWORD Z'14000000'
434  $MAIN$:
435  .MAIN.:
436                     $PSECT .DATA.,MD ,CAT
437  stack:             $WORD Z'FFFFFFFFFFFFFFFF':16384
438                     $END
```

Line 53: The P-code label, 1 3, shown as a comment on line 52, is translated to the APAL label 13.

Lines 54 and 55: The lex P-code instruction is the first instruction of a procedure or function. The calling procedure has computed the address of the base of the current stack frame and put the address in the new mark pointer register (nmp). The operand of lex P-code instruction designates which mark pointer register in the static link display to put the new mark pointer. The mark pointer register is one less than the operand of the lex P-code instruction.

Lines 58 through 66: The Vector Pascal statement on line 13 of Table 50

```
pivotrow := 3;
```

is translated into APAL statements on line 58 through 66.

Lines 56 and 57: No APAL statements are generated by entry (ent) P-code instructions.

Lines 58 through 60: Data pad register dpx(-8) has been allocated to receive the integer constant 3. The constant is the operand in the load constant integer (ldci) P-code instruction shown on line 58 (the operand has been truncated). Only registers dpx(0) and dpy(0) can receive a constant as shown on line 59. Register dpx(0) can be referred to as dpx and dpy can be used wherever dpy(0) occurs.

Line 61: The check P-code instruction is disabled and no code is produced for it.

Lines 62 through 66: The P-code instruction is store at base-level address 72. Address 72 is the address of variable pivotrow. The APAL statements store the constant, 3, in the integer location starting at 72 bytes from the base mark pointer. The relative address, 72, is stored in address register sp(41). The base mark pointer, mp0, is added to the relative address on line 64. The sum is shifted right two positions (line 64). This has the effect of dividing by four. The sum needs to be divided by eight to transform the byte address into a word address. The next APAL statement (line 65) shifts the sum again, completing the transformation, and sets the Main Memory address register. The APAL statement on line 66 asserts the Main Memory address again and places the constant in register dpx(-8) on the data lines. Three instruction cycles later (line 72) the value in dpx(-8) will be copied into the designated Main Memory location. Registers dpx(-8) and sp(41) are returned to the pool of available registers.

Lines 67 through 75: The statement

```
c := 3;
```

on line 14 in Table 50 is translated to APAL statements on lines 67 through 75. The sequence of APAL statements which implement this Vector Pascal statement are identical to those explained on lines 58 through 66. The only difference is the address of variable c (80) whereas the address of variable pivotrow is 72.

Lines 76 through 95: The loop variable, r, is initialized to the value of variable pivotrow plus one.

Lines 76 through 83: The absolute address of variable pivotrow is computed on lines 77 through 80. The relative address of variable pivotrow, 72, is added to the address of the base mark pointer (mp0). The sum is the absolute address. The absolute address is divided by eight to transform it to a word address from a byte address. On line 80 the Main Memory address register is set. Three instruction cycles later, on line 83, register dpx(-8) receives the value of variable pivotrow. No work is performed by the no operation (nop) APAL statements on lines 81 and 82. The nop statements are inserted to wait for Main Memory to deliver the value of variable pivotrow.

Lines 84 through 86: The constant one is assigned to register dpy(-8).

Lines 87 through 90: The value of variable pivotrow in incremented by one. Register dpx(-8) contains the value of variable pivotrow. Register dpy(-8) contains the constant one. The add integer operation (fiadd) on line 88 directs the floating point adder to begin an integer addition. The addition process continues on line 89 and the sum is placed in register dpy(-8) on line 90. Register dpx(-8) is returned to the pool of available registers.

Lines 91 through 95: The loop variable, r, is assigned the starting value, pivotrow + 1 which is computed on lines 76 through 95.

Lines 96 through 104: The Vector Pascal compiler allocates a location (address 904) to deposit the loop limit. The loop limit is equal to the constant rank which is defined on line 3 of Table 50. Constant rank is equal to 10. The constant 10 is put into register dpx(-8) on line 99 and, subsequently, put into the loop limit on line 104.

Line 105: Label 16 marks the top of the for-loop listed on lines 15 through 17 in Table 50.

Lines 106 through 133: The loop variable, r, is compared against the loop limit in address 904. If the loop variable is less than or equal to the loop limit, the loop is executed one more time. If the loop variable is greater than the loop limit then control passes to APAL statements following label 17 on line 386.

Lines 106 through 113: Fetch the value of variable r and put it in register dpx(-8).

Lines 114 through 121: Fetch the value of the loop limit and put it register dpy(-8).

Lines 122 through 130: Compare the loop variable against the loop limit using the floating point adder. The result of the comparison is available three instructions after the comparison is initiated. If the loop variable is less than or equal to the loop limit then transfer control to label t0. Label t0 is the true-branch and label f0 is the false-branch. A true-value is put in register sp(41) when the true-branch is executed and, similarly, a false-value is put in register sp(41) when the false branch is executed. A true-value is equal to one and a false-value is equal to zero.

Lines 131 through 133: If the result of the previous comparison is false then control is transferred to label 17.

Lines 134 through 185: The vector descriptor of the vector on the left hand side of the Vector Pascal statement

```
A[r,c+1..rank] := A[r,c+1..rank] +
```

A[r,c]*A[pivotrow,c+1..rank];

is computed and put in registers. The base address of the vector is put in register sp(42). The low index value is put in register dpy(-8). The value of the high index is put in register dpx(-8). The zero-bias is put in register dpx(-7) and the stride is put in register dpy(-7).

Lines 135 through 137: Fetch the base address of matrix A and put it in register sp(42).

Lines 138 through 144: Fetch the value of the row index, r, and put it in register dpx(-8).

Lines 146 through 150: Decrement the row index to remove the bias.

Lines 151 through 159: Compute the base address of row r in matrix A. Find the offset to row r by multiplying the unbiased value of the row index by the number of bytes in a row (80). Add the offset to the base address of matrix A. The APAL statement on line 158 copies the offset from data pad register dpy to address register sp(44).

Lines 160 through 167: Fetch the value of the column index, c, and put it in register dpx(-8).

Lines 168 through 174: Compute the value of the low index in the vector descriptor by incrementing the column index.

Lines 175 through 177: Put the value of the high index in register dpx(-8).

Lines 178 through 180: Put the zero-bias value in register dpx(-7).

Lines 181 through 183: Put the value of the stride in register dpy(-7).

Lines 184 and 185: No code is produced for the roll up (rlu) and load vector (ldv) instructions.

Lines 186 through 237: The vector descriptor of the vector directly to the right of the assignment token (:=) in the Vector Pascal statement

```
A[r,c+1..rank] := A[r,c+1..rank] +
                  A[r,c]*A[pivotrow,c+1..rank];
```

is computed and put in registers. The base address of the vector is put in register sp(43). The low index value is put in register dpy(-6). The value of the high index is put in register dpx(-6). The zero-bias is put in register dpx(-5) and the stride is put in register dpy(-5). APAL statements which create the vector descriptor are identical, except for the registers, to APAL statements on lines 134 through 185. Refer to the paragraphs above for a detailed explanation.

Lines 238 through 293: The value of the multiplier, A[r,c] is read from Main Memory and put into register dpx(4).

Lines 238 through 240: The base address of matrix A is put in register sp(44).

Lines 241 through 248: The value of the row index, r, is put in register dpx(4).

Lines 250 through 254: The bias is subtracted from the row index.

Lines 255 through 263: The base address of matrix A is replaced by the base address of row r in matrix A. Register sp(44) contains the base address of row r. Refer the paragraph describing lines 151 through 159 for a detailed explanation of this process.

Lines 264 to 277: The column index, c, is put in register dpx(4) and the bias is subsequently removed. This process is identical to that performed on the row index.

Lines 278 through 286: The address of A[r,c] is found and put in register sp(44). Given the base address of row r in register sp(44), the offset to column c is added to the base address, yielding the address of A[r,c].

Lines 287 through 293: The value of A[r,c] is put in register dpx(4).

Lines 290 through 344: The vector descriptor of the pivot row vector in the Vector Pascal statement

```
A[r,c+1..rank] := A[r,c+1..rank] +
                  A[r,c]*A[pivotrow,c+1..rank];
```

is computed and put in registers. The base address of the vector is put in register sp(44). The low index value is put in register dpx(5). The value of the high index is put in register dpy(4). The zero-bias is put in register dpy(5) and the stride is put in register dpx(6). APAL statements which create the vector descriptor are identical, except for the registers, to APAL statements on lines 134 through 185. Refer to the paragraphs above for a detailed explanation.

Lines 346 through 365: No APAL statements are produced for the P-code instructions multiply scalar vector (msv) and add vector (adv). The store vector instruction (stv) is translated in this instance to APAL statements which copy the three vector descriptor components to the registers which serve as input arguments to the multiply-add (ma) run-time library procedure. The multiply-add procedure is called on line 365. A listing of the multiply-add procedure is contained in Table 53. The multiply-add procedure exploits the parallelism of the FPS-164.

Lines 366 through 385: Loop variable r is incremented and control is transferred to the loop test beginning on line 105.

Lines 388 through 398: Values in the static and dynamic link positions in the current stack mark are fetched. The dynamic link will be used to restore the mark pointer identified by the static link. The dynamic link is put in register sp(1). The static link is put in the Table memory address register which serves as the offset for an indexed jump. The jump statement on line 396 transfers control to the location of label rstrmp (restore mark pointer) plus the offset in the Table memory address

register. The jump statement always transfers control to one of the move statements (mov) on lines 415 to 430. In this case mark pointer zero (mp0) is restored and the value of the static link is zero. APAL statements on lines 415 and 416 are executed and control is returned to the caller.

Lines 399 through 412: Execution begins on line 399. APAL statements on lines 401 through 410 create the base stack mark. The subroutine-jump statement on line 412 transfers control to label 13 on line 53. Line 53 marks the beginning of the body of program E2.

Line 414: Control is returned to SUM.

Lines 415 through 430: Refer to the discussion of lines 388 through 398.

Lines 431 through 433 contain APAL directives which allow this program to execute under the control of SJE and SUM.

Lines 434 through 438: APAL statements on these lines define storage for the stack.

The reader is invited to consider the run-time library routine, MA, listed in Table 53. Routine MA receives four operands, a descriptor of the output vector, two input vector descriptors, and a scalar. Routine MA multiplies one input vector by a scalar, adds the other vector to the product, and assign the sum to the output vector.

Routine MA is written in Array Processor Assembler Language and exploits the parallelism provided by the FPS-164 instruction set. Each line of the listing in Table 53 is described below.

Lines 1 through 32: The title directive on line 1 must appear on the first line of an APAL routine. It associates the name (ma) with a brief comment (multiply add) describing the function of the routine. Symbolic names for registers and constants are defined in the files included by the insert directives on lines 2 through 4. Lines 6 through 32

define input parameters and local variables which are discussed in detail in the following paragraphs.

Table 53. Run-time library routine MA

```
Line APAL Statement
1              $TITLE ma,'multiply add'
2              $insert'=PS.KEY'
3              $insert'=SYS.KEY'
4              $insert'=TM.KEY'
5
6     SAdr1    $equ sp(1)          "Base address of vector 1
7     SAdr2    $equ sp(2)          "Base address of vector 2
8     DAdr     $equ sp(0)          "Base address of dst vector
9     DpStep1  $equ dpx(-4)        "Step size of vector 1
10    DpStep2  $equ dpx(-3)        "Step size of vector 2
11    DpStepD  $equ dpx(-2)        "Step size of dst vector
12    Lo1      $equ dpx(-1)        "Min index value for vector 1
13    Lo2      $equ dpx( 1)        "Min index value for vector 2
14    LoD      $equ dpx( 2)        "Min index value for vector 2
15    NdxL1    $equ dpy(-4)        "Bias index of vector 1
16    NdxL2    $equ dpy(-3)        "Bias index of vector 2
17    NdxLD    $equ dpy(-2)        "Bias index of dst vector
18    Hi1      $equ dpy(-1)        "Max index value for vector 1
19    Hi2      $equ dpy( 0)        "Max index value for vector 2
20    HiD      $equ dpy( 1)        "Max index value for dst vect
21    k        $equ dpx( 0)
22    cnt      $equ dpx( 3)        "Hi1 - Lo1
23    re       $equ dpy( 2)
24    DpOfs1   $equ dpy( 2)
25    DpOfs2   $equ dpy(-3)
26    DpOfsD   $equ dpy(-2)
27    Ofs1     $equ sp(3)
28    Ofs2     $equ sp(4)
29    OfsD     $equ sp(5)
30    Step1    $equ sp(6)
31    Step2    $equ sp(7)
32    StepD    $equ sp(8)
33             $ENTRY ma
34             $PSECT .CODE.,PS ,CAT
35             $PENTF ma
36             flshi -3,DpStep1
37             flshi -3,DpStep2
38             flshi -3,DpStepD;DpStep1<fa
39             fapush           ;DpStep2<fa
40                               DpStepD<fa
41             ldtma; db=tm$i1
42             fisub Lo1,NdxL1; movrr SAdr1,SAdr1
```

Table 53. Continued

```
Line APAL Statement
43            fisub Lo2,NdxL2; movrr SAdr2,SAdr2
44            fisub LoD,NdxLD; movrr DAdr ,DAdr ; DpOfs1<fa
45            fisubr tm,DpOfs1;movr  SAdr1,SAdr1; DpOfs2<fa
46            fisubr tm,DpOfs2;movr  SAdr2,SAdr2; DpOfsD<fa
47            fisubr tm,DpOfsD; fimul  DpStep1,fa; movr DAdr
48            fapush          ; fimul  DpStep2,fa
49                              fimul  DpStepD,fa
50            fisub Hi1,Lo1   ; fmpush; DpOfs1<fm
51            fapush          ; fmpush; DpOfs2<fm
52            cnt<fa          ;         DpOfsD<fm
53            ldspi Ofs1; db=DpOfs1
54            ldspi Ofs2; db=DpOfs2
55            ldspi OfsD; db=DpOfsD
56            ldspi Step1;db=DpStep1
57            ldspi Step2;db=DpStep2
58            ldspi StepD;db=DpStepD
59            add Ofs1,SAdr1
60            add Ofs2,SAdr2
61            add OfsD,DAdr
62
63            add Step1,SAdr1;setma
64            add Step2,SAdr2;setma
65            nop
66
67            add Step1,SAdr1;setma;fmul k,md
68            add Step2,SAdr2;setma;fmpush;re<md
69            nop                   ;fmpush      ;
70               fisubr tm,cnt
71
72    am:     add Step1,SAdr1;setma;fmul k,md   ;
73               fadd fm,re
74            add Step2,SAdr2;setma;fmpush;re<md;
75               fapush;cnt<fa
76            add StepD,DAdr ;setma;fmpush      ;
77              mi<fa;bfge am; fisubr tm,cnt
78
79            $PEXIT ma
80            $PSECT .DATA.,MD ,CAT
81            $END
```

Lines 6 through 8: The base address components of the three vector descriptors arrive in registers sp(0), sp(1), and sp(2). The name SAdr is an abbreviation of source address and DAdr is an abbreviation of destination address.

The term source is the same as input and destination is synonymous with output in the discussion of routine MA. The equate directive assigns the alias on the left of the directive to the register on the right of the directive. For example, the name DAdr is an alias for register sp(0).

Lines 9 through 11: The stride components of the three vector descriptors arrive in registers dpx(-4), dpx(-3), and dpx(-2). The stride components in the source vectors are referred to by their aliases, DpStep1 and DpStep2, and the stride component of the destination vector is named DpStepD. A stride is added to the address of the previous element in a vector to compute the address of the next element. Address computations are performed by the address ALU (see Figure 10) using address registers. The mnemonic Dp in the register aliases indicates the stride components are in data pad registers. After converting the stride components to word increments (lines 36 through 40) they are put in address registers (lines 56 through 58) where they can be used in address arithmetic.

Lines 12 through 14: The low index components arrive in registers dpx(-1), dpx(1), and dpx(2). The low index components in the source vectors are referred to by their aliases, Lo1 and Lo2, and the low index component of the destination vector is named LoD. The bias must be subtracted from low index components (lines 42 through 44) before the offsets to the first element in the vectors can be found. The offset to the first element is found by multiplying the unbiased low index component and the stride.

Lines 15 through 17: The zero-bias components arrive in registers dpy(-4), dpy(-3), and dpy(-2). The zero-bias components in the source vectors are referred to by their aliases, NdxL1 and NdxL2, and the zero-bias component of the destination vector is named NdxLD. The bias must be subtracted from low index components (lines 42 through 44)

before the offsets to the first element in the vectors can be found.

Lines 18 through 20: The high index components arrive in registers dpy(-1), dpy(0), and dpy(1). The high index components in the source vectors are referred to by their aliases Hi1 and Hi2, and the high index component of the destination vector is named HiD. A low index component is subtracted from a high index component is used to compute how many elements are in the vectors (lines 50 through 52). The difference is put in a register referred as cnt (line 22).

Line 21: The multiplier used to scale source vector one arrives in register dpx(0) and is referred to by its alias, k.

Line 22: Register cnt (dpx(3)) is equal to one less than the number of elements in the three vectors.

Line 23: Register re (dpy(2)) is used to store elements from source vector two.

Lines 24 through 29: The offsets to the first element of the three vectors are computed by using the floating point multiplier. Data pad registers, DpOfs1, DpOfs2, and DpOfsD, receive the resulting computation. The mnemonic Dp indicates a data pad register. The offsets are then copied into address registers, Ofs1, Ofs2, and OfsD.

Lines 30 through 32: Registers Step1 (sp(6)), Step2 (sp(7)), and Step3 (sp(8)) contain the stride components for the two source vectors and the destination vector. The stride values have been converted from byte to word increments. A stride value is added to the address of the previous element in a vector to compute the address of the next element. All address arithmetic is performed using address registers.

Lines 33 through 35: The entry directive permits this routine to be bound to Vector Pascal programs. The $PSECT

directive defines a program section. The $PENTF directive defines how registers are saved and restored when the control passes to and from the routine.

Routine MA consists of three parts:

1. Loop setup: lines 36 through 61:
2. Filling the pipe: lines 63 through 71
3. The loop: lines 72 through 77

Lines 36 through 61: This section converts input vector descriptors to values that can be readily used by the loop. Stride values must be converted from byte to word increments. Offsets to starting elements must be computed. Actually, since the loop begins by adding the stride to the base address, offsets to one element preceding the starting element must be computed. The biases on low index values must be removed. All addresses must be converted from byte to word addresses.

Lines 36 through 40: The floating point adder is used to shift stride components right three bit-positions. This action divides stride components by eight and converts them from byte increments to word increments. The first shift is initiated on line 36 and completed on line 38. Subsequent shift operations on lines 37 and 38 initiate the second and third shifts, and push previous operations through the floating point adder pipeline. The operation code fapush on line 39 completes the last shift initiated on line 38. Results of the last shift operation are available two instruction cycles after initiation (line 40).

Line 41: The Table Memory address register is set to the address of the integer constant one. Two instruction cycles later (line 43) the constant will be available. The constant one will remain on the Table Memory data lines until another address is assigned to the Table Memory address register. The constant is not referenced until line 45.

Lines 42 through 44: The floating point adder is used in parallel with the address ALU in this section. The floating point adder is used to remove the biases by subtracting them from low index values. The address ALU is used to convert the base address components from byte to word addresses by shifting.

The first difference is initiated on line 42 and completed on line 44. Differences initiated on lines 43, 44, and 45 push previous results through the floating point adder pipeline. Registers DpOfs1, DpOfs2, and DpOfs3 receive the results of the floating adder (fa) on lines 44, 45, and 46 respectively. Registers DpOfs1, DpOfs2, and DpOfs3 now contain the unbiased low index values.

The address ALU is used to shift the two source addresses, SAdr1 and SAdr2, and the destination address, DAdr, right two bit-positions. The effect of this action is to divide the base address components by four. To completely convert address components to word addresses they must be divided by eight. Instructions on the following lines complete the process of converting the base address components.

Lines 45 through 47: The floating point adder and address ALU are used in parallel on lines 45 and 46. All three functional units including the floating point multiplier are used on line 47.

In the first column (columns are separated by semicolons), the floating point adder is used to decrement the low index values. The resulting value, when multiplied by its stride, will produce an offset to the element directly preceding the first element. The Table Memory (tm) data lines contain the constant one used to decrement the low index values. Low index values are called offsets (DpOfs1, DpOfs2, and DpOfsD) and, indeed, they are being converted to offsets.

In the second column on lines 45 and 46, conversion of source addresses, SAdr1 and SAdr2, is completed. Conversion of the destination address, DAdr, is completed in the third column on line 47.

The floating point multiplier (line 47, column 2) is used to start the product of an unbiased low index value (fa) and the corresponding stride (DpStep1). Register DpOfs1 has been decremented by the floating point adder. The difference is in the output of the floating point adder (fa). The product will be the offset to the element preceding the first element. The output of the floating point adder is directly connected to the right hand input of the floating point multiplier.

Lines 48 and 49: The floating point multiplier is used to compute offsets to the elements preceding the first element of source vector two and the destination vector. The products are available three instructions after they are initiated. Offsets DpOfs1, DpOfs2, and DpOfsD, are copied from the output of the floating point multiplier (fm) in column three on lines 50 through 52.

The floating point multiplier must be executed in three instruction cycles before the product initiated in the first cycle is formed. New products may be initiated on any cycle. The fmpush instruction causes the floating point multiplier to execute without initiating a multiplication so products started on previous cycles can be completed.

Lines 50 through 52: The floating point adder is used to compute the number of elements in a vector (see column 1). The number of elements is assigned to register cnt (count). The count is computed by subtracting the low index value from the high index value. Actually, since the vector includes both the low index and the high index values, the count is one less than it should be. However, the loop test on line 77 compensates for that deficiency.

Lines 53 through 58: Offset and stride values are copied from data pad registers into address registers.

Lines 60 through 62: Offsets are added to base addresses. Resulting sums are the addresses of elements directly preceding the element indexed by the low indexes. The address ALU is used to perform the sums.

Lines 63 through 71: This section contains instructions which fill the loop pipeline. The function of the loop is to multiply an element of source vector one by the multiplier, k, add the product to an element of source vector two, and assign the sum to an element in the destination vector. Every time the loop is executed one more element is assigned. However, partial results for subsequent elements are also computed. The loop depends on these partial results to produce the final result which is put in the destination vector. Therefore, partial results must be computed in advance of the actual loop instructions. The statements on lines 63 through 70 perform that function.

Lines 63 through 65: The first task to perform is to fetch the source operands. Main Memory read operations are initiated to fetch elements of the two source vectors.

Line 63: The address of the first element of source vector one is computed (add Step1, Sadr1) and assigned to the Main Memory address register (setma). Three instruction cycles later (line 67) the value of the element will be available.

Line 64: The address of the first element of source vector two is computed (add Step2, Sadr2) and assigned to the Main Memory address register (setma). Three instruction cycles later (line 68) the value of the element will be available.

Line 65: Wait for Main Memory to deliver data.

Lines 67 through 70: In the left column the instructions on lines 63 through 65 are repeated to start

reading the second elements of the source vectors. On line 67 the operation to scale the first element of source vector one is initiated (fmul k,md). The mnemonic md refers to the Main Memory data lines on which the first element of source vector one is present. On line 68, the first element of source vector two is retained in register re (re<md) and the multiplication started on the previous line is continued (fmpush).

Lines 69 and 70 constitute a single instruction cycle. The multiply operation started on line 67 is continued (fmpush) and the count (cnt) is decremented by one (fisubr tm,cnt). Actually, since the floating point adder is employed to decrement the count, the process is not complete. Recall that the Table Memory (tm) data lines still retain the constant one.

Lines 72 through 77: This loop consists of three instruction cycles. Lines 72 and 73 are one cycle. Lines 74 and 75 are one cycle and lines 76 and 77 are one cycle.

Lines 72 and 73: The address ALU is used to start fetching another element of source vector one (add Step1,SAdr1;setma). The floating point multiplier is used to start scaling the previous element of source vector one by the multiplier, k (fmul k,md). The product of the previous multiply operation is available and added to the corresponding element of source vector two (fadd fm,re). The floating point adder is used to start the foregoing addition. Note that all three functional units are used.

Lines 74 and 75: The address ALU is used to start fetching another element of source vector two (add Step2,SAdr2;setma). The multiplication started on the previous cycle is continued (fmpush) and the addition started on the previous cycle is continued (fapush). Another element from source vector two is copied from the Main Memory data lines to register re (re<md). The adder

has decremented the loop count and it is saved in register cnt (cnt<fa). The adder and Main Memory produce new results every cycle. These results are needed in subsequent computations and must be retained in registers.

Lines 76 and 77: The address ALU is used to compute the address of an element in the destination vector (add StepD,DAdr). The element address is assigned to the Main Memory address register (setma). The result of the sum started at the beginning of the loop is assigned to the Main Memory data lines (mi<fa). The sum assigned to Main Memory is the sum of an element of source vector two and the product of the scale factor, k, and the corresponding element of vector one. The floating point multiplier is directed to continue (fmpush). The floating point adder is used to start decrementing the loop count (fisubr tm,cnt). The floating point adder updates a status register three instruction cycles after an operation is started. The status register contains flags describing the result of subtraction. If the loop count is greater than or equal to zero then control is transferred to the start of the loop (bfge am); otherwise, the loop terminates and control is returned to the Vector Pascal program calling routine MA.

6. RESULTS

This chapter presents the results of executing the Standard Pascal and Vector Pascal versions of LINPACK discussed in chapter 4.

6.1 Measurement

The procedures which constitute the LINPACK benchmark are not a program. The LINPACK procedures must be included in a program which generates a matrix and which measures execution time. Appendix A contains a listing of the program that manages the Standard Pascal version of LINPACK and appendix B contains a listing of the program that manages the Vector Pascal version of LINPACK. The programs in appendices A and B will be referred to as the Standard Pascal version of LINPACK and the Vector Pascal version of LINPACK respectively in this chapter.

The Standard Pascal version of LINPACK and the Vector Pascal version of LINPACK were translated and executed using the procedure discussed in section 5.1. The Vector Pascal compiler was used to translate both versions of LINPACK.

The Vector Pascal compiler is derived from the P4 Standard Pascal compiler described by Pemberton and Daniels [3]. The Vector Pascal compiler produces the same P-code representation of the Standard Pascal version of LINPACK as the P4 compiler. The Vector Pascal compiler is, therefore, equivalent to a Standard Pascal compiler for Standard Pascal programs.

Each version of the LINPACK program was compiled and executed ten times. The matrix size was incremented by five in subsequent compilations. Results in Table 54 show how much time was required for the Standard Pascal and Vector Pascal versions of LINPACK to solve matrices having a rank of five to a rank of fifty. Speedup of the Vector Pascal version of LINPACK over the the Standard Pascal version of

LINPACK is also shown in Table 54. Speedup is defined to be the ratio of the Standard Pascal version of LINPACK divided by the Vector Pascal version of LINPACK. A graph of speedup as a function of matrix rank is shown in Figure 13.

Table 54. LINPACK execution times

Rank of Matrix	Standard Pascal version of LINPACK (microseconds)	Matrix Pascal version of LINPACK (microseconds)	Speedup
5	3,659	2,028	1.8042
10	18,600	6,358	2.9254
15	51,242	12,857	3.9855
20	107,557	21,836	4.9257
25	193,229	32,374	5.9686
30	317,880	46,336	6.8603
35	482,575	62,522	7.7185
40	703,619	84,810	8.2964
45	975,650	104,429	9.3427
50	1,309,409	130,199	10.0570

The measurements listed in Table 54 were made by reading the clock on the FPS-164 Scientific Computer. The standard function clock was added to the Vector Pascal compiler. Standard function clock returns the number of microseconds elapsed since the executing task was initiated. The clock is read directly before and directly after the invocation of procedures DGEFA and DGESL. The execution times of procedures DGEFA and DGESL are recorded in variables DGEFAtime and DGESLtime. Variables DGEFAtime and DGESLtime are added to produce the measurement shown in Table 54.

This technique produces marginally inaccurate data since the time required to read the clock is included in the measurement. The time required to read the clock is, at most, a few microseconds. The smallest measurement is three orders of magnitude greater than the time needed to read the clock.

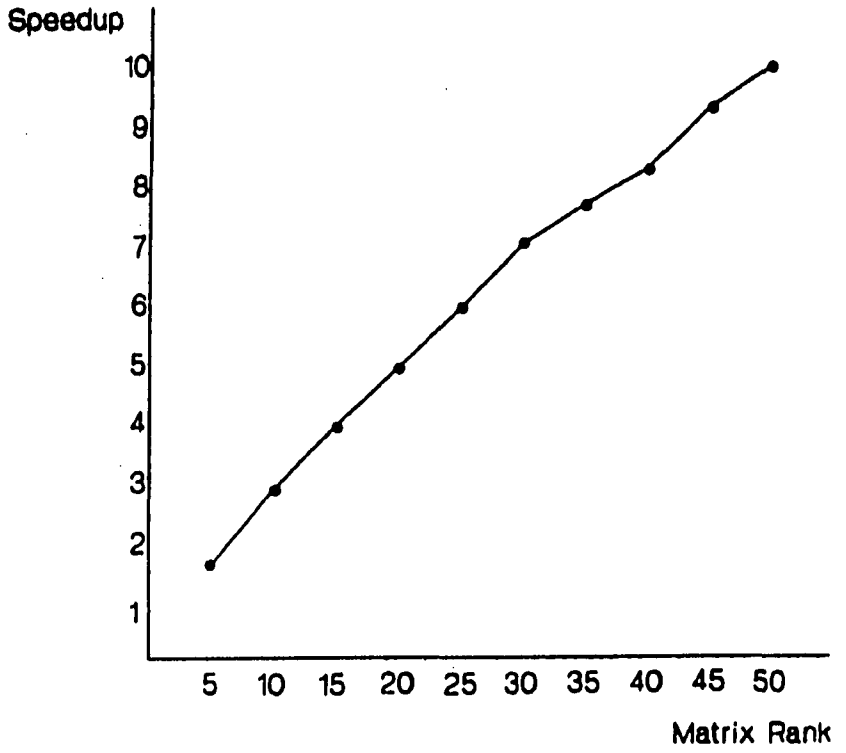


Figure 13. Speedup versus matrix rank

6.2 Analysis

It is possible to derive an equation for the speedup listed in Table 54 by counting the number of statements executed in both versions of LINPACK. The number of APAL statements executed in the Standard Pascal version of procedures DGEFA and DGESL is listed beside corresponding line numbers in Tables 55 and 56 respectively. The line numbers refer to Standard Pascal statements listed in Tables 34 (Standard Pascal Procedure DGEFA) and 35 (Standard Pascal Procedure DGESL). The number of APAL statements is expressed in terms of the rank of the input matrix, n, and includes the number of statements executed in any procedure called.

Table 55. Standard Pascal execution cost for DGEFA

DGEFA Line (Table 34)	Number of APAL Statements Executed
10	1
11	11
12	17 + 21(n-1)
13	11(n-1)
14	167(n-1) + 138n(n-1)/2
15	35(n-1)
16	65(n-1)
17	0
18	(n-1)
19	0
20	21(n-1)
21-26	33(n-1) + 228(n-1)(n+2)/2
27	11(n-1)
28	96(n-1)
29	87(n-1) + 145n(n-1)/2
30-31	38(n-1) + 174n(n-1)/2 + 143n(n-1)(2n-1)/6
32	0
33	16(n-1)
34	66
35	25
36	10
Total = $47.67n^3 + 271n^2 + 511.33n - 700$	

The total number of APAL instructions executed in the Standard Pascal version of LINPACK is:

$$47.67n^3 + 431.5n^2 + 817.83n - 665$$

Tables 57 and 58 list the number of APAL instructions executed beside each line of the Vector Pascal version of LINPACK. Line numbers in Table 58 refer to line numbers for procedure DGEFA listed in Table 41 and line numbers in Table 59 refer to line numbers listed beside the statements of procedure DGESL in Table 42.

The total number of APAL instructions executed in the Vector Pascal version of LINPACK is:

$$n^3 + 198.5n^2 + 1371.5n - 1419$$

Table 56. Standard Pascal execution cost for DGESL

DGESL Line (Table 35)	Number of APAL Statements Executed
11	1
12	12 + 21n
13	0
14	36n
15	21n
16	0
17-19	131n
20	0
21	22n + 21n(n-1)/2
22	164n(n-1)/2
23	16n
24	12 + 21n
25	0
26	36n
27	22n + 21n(n+1)/2
28	99n(n-1)/2 + 16n(n+1)/2
29	88n
30	16n
31	10
	Total = $160.5n^2 + 306.5n + 35$

Dividing the Standard Pascal execution cost for LINPACK by the Vector Pascal execution cost produces an expression for speedup (see Table 59). Values for the speedup equation shown in Table 59 are listed in Table 60.

As the matrix rank, n, approaches infinity the speedup approaches 47.67, the coefficient of n^3 in the numerator.

A comparison of the speedup values computed in Table 62 with those recorded in Table 54 reveals a slight difference. The differences can be accounted for in the assumptions that were made to derive the expression for speedup. It was assumed that the initial matrix was badly formed so that the longest branch of every test was executed. This assumption tends to improve the expression for speedup because vector features were used to replace Standard Pascal statements on some segments of the program. Another assumption is that each APAL statement requires the same time to execute (187

ns). This assumption is not always true. Two consecutive references to the same memory bank will delay execution by the latency of that memory. The Vector Pascal version of LINPACK executes a little slower than the prediction in Table 61 because of the delay due to consecutive memory references. If the expression in Table 60 were increased somewhat then the speedup would be decreased.

Table 57. Vector Pascal execution cost for DGEFA

DGEFA Line (Table 41)	Number of APAL Statements Executed
10	1
11	11
12	17 + 21(n-1)
13	11(n-1)
14	201(n-1) + 130n(n-1)/2
15	35(n-1)
16	65(n-1)
17	0
18	(n-1)
19	0
20	21(n-1)
21	108(n-1) + 5n(n-1)/2
22	11(n-1)
23	96(n-1)
24-25	132(n-1) + 2n(n-1)/2
26-28	22(n-1) + 258n(n-1)/2 + 3n(n-1)(2n-1)/6
29	0
30	16(n-1)
31	66
32	25
33	10
Total = $n^3 + 196n^2 + 542n - 610$	

Table 58. Vector Pascal execution cost for DGESL

DGESL Line (Table 42)	Number of APAL Statements Executed
11	1
12	12 + 21(n-1)
13	0
14	36(n-1)
15	21(n-1)
16	0
17-19	131(n-1)
20	0
21	16(n-1)
22	150(n-1) 3n(n-1)/2
23	16(n-1)
24	123
25-27	318(n-1) + 2n(n-1)/2
28	10
	Total = $2.5n^2 + 829.5n - 809$

Table 59. LINPACK equation for speedup

$$\text{Speedup} = \frac{47.67n^3 + 431.5n^2 + 817.83n - 665}{n^3 + 198.5n^2 + 1371.5n - 1419}$$

Table 60. LINPACK speedup values

Rank of Matrix	Measured Speedup	Computed Speedup
5	1.80	1.92
10	2.93	2.97
15	3.99	4.01
20	4.93	5.02
25	5.97	5.99
30	6.86	6.93
35	7.72	7.82
40	8.30	8.67
45	9.34	9.49
50	10.05	10.28

7. CONCLUSIONS

Significant contributions presented in this dissertation include:

i the Vector Pascal Language

ii the transcription of LINPACK procedures into Vector Pascal illustrating the use of the language.

iii evidence suggesting that language design can contribute to the performance of programs which use vector operations

7.1 The Vector Pascal Language

One of the goals of this investigation is to discover and find solutions for the problems associated with defining and translating vector expressions. The survey of current literature in chapter 2 indicates that these problems are under investigation. Certainly, those participating in the definition and implementation of Actus and Fortran 8X must address the problems of vector expressions. The Vector Pascal programming language is a vehicle for studying the problem of vector expressions.

The essential issues surrounding the problem of vector expressions are addressed in this dissertation. The Standard Pascal grammar was modified to support vector expressions. An existing compiler was modified to accept the altered grammar. New P-code instructions were defined to support vector expressions defined in the grammar. A code generator was designed to efficiently translate vector P-code instructions into corresponding APAL statements. The definition of Vector Pascal is an example of how vector expressions are added to an existing language without requiring an entirely new compiler. Vector expressions in Vector Pascal are also an abstraction of the underlying computer architecture.

7.2 LINPACK Transcriptions

One of the goals of this study is to design language features which support applications based on the principles of linear algebra. Finding the solution to a system of linear equations is, perhaps, the most widely used application resulting from the study of linear algebra. LINPACK procedures DGEFA and DGESL factor and solve a system of linear equations. Vector features defined in chapter 3 are used in the Vector Pascal version of LINPACK presented in chapter 4. An application based in linear algebra can, therefore, be expressed using the syntax of Vector Pascal. Support for an application of linear algebra is manifested by the performance benefits described in chapter 6.

Vector Pascal is not designed to support all applications of linear algebra. However, the features included in Vector Pascal, open array parameters, slices, and standard vector operations, are sufficient to test this thesis. The LINPACK program employs all these features. Function IMAX is called using an open array parameter. Vector operations on row and column slices replace procedures DAXPY and DSCAL. Vector operations include scaling, addition, and inner product.

7.3 Speedup

A question posed in this dissertation is: can the performance of a computation-intense program be improved by altering it to take advantage of the vector operations available in a language that contains support for vector operations? Results in chapter 6 suggest that performance can be improved.

The Vector Pascal version LINPACK requires less time to execute than the Standard Pascal version primarily because of loop optimizations. Standard Pascal contains no support for vector operations and, as a result, vector operations

must be constructed from scalar operations on individual elements using loops.

Compilers usually translate one statement into many machine instructions. It is difficult for a compiler to recognize a vector operation that has been coded as a loop. A compiler has difficulty translating the loop management, array element dereferencing, and loop computations into an integrated sequence of instructions which perform a vector operation. Loop management, element dereferencing, and loop computations are usually translated into separate code segments. A FPS-164 computer is most efficiently utilized if the loop management, element dereferencing, and loop computations are packed into an integrated sequence of APAL instruction. The discussion of run-time library routine MA in chapter 5 is an example of how vector addition and scaling can be packed into a sequence of instructions which, after initialization, require only three instructions per element. This should be compared to over 100 APAL instructions produced for procedure DAXPY which performs the same function as routine MA. Since procedure DAXPY and routine MA are executed $O(n^2)$ times the speedup of routine MA over procedure DAXPY is significant.

Since Vector Pascal translates vector operations into APAL loops standard loop optimizations are employed to improve performance. Dereferencing an element of a matrix in Standard Pascal is performed in the following way. The low bound is subtracted from the row index. The difference is multiplied by amount of memory occupied by a row. The product is the row-offset. The row-offset is added to the base address of the matrix. The sum is the base address of the row. The low bound is subtracted from the column index. The difference is multiplied by the amount of memory occupied by an element of the matrix. The product is the column-offset. The column-offset is added to the base

address of the row. The resulting address is used to fetch the referenced element. The cost to dereference one element of a matrix is two subtractions, two additions, and two multiplications.

If it is known that neighboring elements in a row or column are going to be accessed in sequence then the cost of dereferencing can be reduced. Once the address of the first element is computed, the remaining elements are a fixed distance from the first. The stride is added to the first element to obtain the second. The cost has been reduced from two subtractions, two additions, and two multiplications to one addition.

Another optimization employed by the Vector Pascal compiler on vector operations is decrementing the loop variable rather than incrementing the loop variable. Compilers are obligated to increment the loop variable when translating a for-loop. In a vector operation the loop variable is implied. As a result the loop variable can be decremented rather than incremented. The FPS-164 has instructions which transfer control to the start of the loop as long as the loop variable is greater than zero. Because of this feature it is advantageous to decrement rather than to increment. If incrementing were employed then the loop limit would have to be subtracted from the loop variable every time the loop was executed. Decrementing eliminates loading and testing the loop limit against the loop variable.

In addition to the foregoing optimizations vector operations are also pipelined and use all the functional units of the FPS-164 simultaneously.

One cannot conclude that the performance of a program will always be improved based on the results in chapter 6. Defining vector operations in a language permits the compiler writer to generate efficient code for vector

operations. However, optimizing compilers generate efficient code for do-loops which are equivalent to vector operations.

7.4 Future Work

Matrix operations could be implemented in subsequent research. The syntax supporting matrix operations is defined in chapter 3. Perhaps additional syntax could be defined. Many algorithms employ expressions for matrix partitions. The list of matrix operations in chapter 3 is not exhaustive. Transpose and determinant functions are well known in linear algebra.

A matrix descriptor similar to the vector descriptor needs to be defined. Additional P-code instructions which support matrix operations need to be defined. The matrix P-code instructions and matrix descriptor need to be integrated with the existing Vector Pascal compiler.

A strategy for handling matrix temporaries needs to be developed. Matrix operations do not have the same property as vector temporaries. Storage for the matrix temporaries must be allocated.

Another topic for future work is to determine if the performance of programs can be improved by using vector operations when the original program was translated using an optimizing compiler.

Another area of research suggested by this dissertation is translating vector expressions. The approach adopted in this investigation is: optimized routines are constructed for every possible vector expression. A variation of this approach has been adopted by Floating Point Systems, Inc. Four volumes of optimized subroutines that perform vector and matrix operations can be purchased from FPS. However, the author hopes that an algorithm to translate arbitrary

vector expressions into highly efficient code can be discovered through additional research.

8. BIBLIOGRAPHY

1. Dongarra, J. J.; Moler, C. B.; Bunch, J. R.; and Stewart, G. W. LINPACK Users' Guide. Philadelphia: Society for Industrial and Applied Mathematics (SIAM), 1979.

2. Floating Point Systems. APFTN64 User's Guide. Beverton, OR: Floating Point Systems, Inc., 1985.

3. Pemberton, S.; and Daniels M. C. Pascal Implementation. New York: Halsted Press, A division of John Wiley & Sons, 1982.

4. Hwang, K.; and Briggs, F. A. Computer Architecture and Parallel Processing. New York: McGraw-Hill, Inc. 1984.

5. Barnes, G. H.; Brown, R. M.; Kato, M.; Kuck, D. J.; Slotnick, D. L.; and Stokes, R. A. "The ILLIAC IV Computer." IEEE Transactions on Computers, C-17, Number 8 (Aug 1968): 746-757.

6. Flynn, M. J. "Very High-Speed Computing Systems." Proceedings of the IEEE, 54, Number 12 (Dec 1966): 1901-1909.

7. Jones, A. K.; Chansler, Jr., R. J.; Durham, I.; Feiler, P.; and Schwans, K. "Software Management of CM* - A Distributed Multiprocessor." AFIPS Conference Proceedings, 46 (1977): 657-663.

8. Swan, R. J.; Fuller, S. H.; and Siewiorek, D. P. "Software Management of CM* - A Modular Multiprocessor." AFIPS Conference Proceedings, 46 (1977): 637-644.

9. Sammet, J. E. Programming Languages: History and Fundamentals. Englewood Cliffs, NJ: Prentice-Hall, 1969.

10. Jensen, K.; and Wirth, N. PASCAL User Manual and Report. 2nd ed. New York: Springer-Verlag, 1974.

11. Brinch-Hansen, P. The Architecture of Concurrent Programs. Englewood Cliffs, NJ: Prentice-Hall, 1977.

12. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Commnunications of the ACM, 21, Number 8 (Aug. 1978): 613-641.

13. Reid, J. K.; and Wilson, A. "The Array Features in Fortran 8X with Examples of Their Use." Computer Physics Communications, 37 (1985): 125-132.

14. Perrott, R. H. "A Language for Array and Vector Processors." ACM Transactions on Programming Langauages and Systems (TOPLAS), 1, Number 2 (Oct. 1979): 177-195.

15. Stevens, K. G. "CFD - A Fortran-like language for the ILLIAC IV." ACM SIGPLAN Notices, 10, Number 3 (1975): 72-80.

16. ICL. DAP: Introduction to Fortran Programming. ICL Technical Publication 6755, 1979.

17. Perrott, R. H.; and Zarea-Aliabadi, A. "Supercomputer Languages." Computing Surveys, 18, Number 1 (Mar. 1986): 5-22.

18. Kuck, D. J.; Kuhn, R. H.; Padua, D. A.; Leasure, B.; and Wolfe, M. "Dependence Graphs and Compiler Optimizations." Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL), Williamsburg, VA, (Jan. 1981): 207-218.

19. Kennedy, K. "Automatic Translation of Fortran Programs to Vector Form." Technical Report 476-029-4. Rice University, Houston TX, October 1980.

20. Scarborough, R. G.; and Kolsky, H. G. "A vectorizing Fortran compiler." IBM Journal of Research and Development, 30, Number 2 (Mar. 1986): 163-271.

21. Siewiorek, D. P.; Bell, C. G.; and Newell, A. Computer Structures: Principles and Examples. New York: McGraw-Hill Book Company, 1982.

22. Wichmann, B. A. Algol 60 Compilation and Assessment. New York: Anderson Press, 1973.

23. Fuller, S. H.; Shaman, P.; Lamb, D.; and Burr, W. E. "Evaluation of Computer Architectures via Test Programs." AFIPS Conference Proceedings, 46 (1977): 147-160.

24. Lucas, H. C. "Performance Evaluation and Monitoring." Computing Surveys, 3, Number 3 (1971): 79-91.

25. Gries, D. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.

26. Aho, A. V.; and Ullman, J. D. Principles of Compiler Design. Reading, MA: Addison-Wesley Publishing Company, 1979.

27. Pyster, A. B. Compiler Design and Construction. New York: Van Nostrand Reinhold Company, 1980.

28. Pratt, T. W. Programming Languages: Design and Implementation. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1975.

29. Barron, D. W., ed. Pascal - The Language and its Implementation. New York: John Wiley & Sons, 1981.

30. Floating Point Systems. APAL64 Programmer's Reference Manual. Beverton, OR: Floating Point Systems, Inc., 1985.

31. Hoffman, K.; and Kunze, R. Linear Algebra. 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1971.

32. Habermann, A.; and Perry, D. Ada for Experienced Programmers. Reading MA: Addison-Wesley Publishing Company, 1983.

33. Wirth, N. Programming in Modula-2. 2nd ed. New York: Springer-Verlag, 1983.

34. Forsythe, G.; and Moler, C. Computer Solution of Linear Algebraic Systems. Englewood Cliffs, NJ: Prentice-Hall, 1967.

35. Floating Point Systems. APAL64 Programmer's Guide. Beverton, OR: Floating Point Systems, Inc., 1985.

36. Knuth, D. E. The Art of Computer Programming. Vol. 1: Fundamental Algorithms. 2nd ed. Reading MA: Addison-Wesley Publishing Company, 1973.

9. APPENDIX A: STANDARD PASCAL VERSION OF LINPACK

```
program lintest;
  const
    partition=4;
    rank=50;
    smallreal = 1.0e-300;
  type
    idxrng  = 1..rank;
    ivector = array[idxrng] of integer;
    rvector = array[idxrng] of real;
    matrix  = array[idxrng] of rvector;
  var
    A: matrix;
    b: rvector;
    ipvt: ivector;
    info: integer;
    job: integer;
    DGEFAtime,DGESLtime : integer;
    mflops: real;

  function uniform: real;
    const
      ulo  =  2.0;
      uhi  = 13.0;
   begin
     uniform := round((uhi-ulo)*random(0) + ulo);
   end uniform ;

  procedure prttitle;
    var c: idxrng;
  begin
    write('  ');
    for c:=1 to partition do
      write(' ','-------',c:1,'-------');
    writeln
  end prttitle ;

  procedure prtivec(var v: ivector);
    var c: idxrng;
  begin
    prttitle;
    write('  ');
    for c:=1 to partition do
      write(' ',v[c]:15);
    writeln;
  end prtivec ;

  procedure prtrvec(var v: rvector);
    var c: idxrng;
  begin
    prttitle;
    write('  ');
    for c:=1 to partition do
      write(' ',v[c]:15);
    writeln;
  end prtrvec ;

  procedure prtmat(var A: matrix);
    var r,c: idxrng;
  begin
```

```
  prttitle;
  for r:=1 to partition do
    begin
      write(' ',r:1);
      for c:=1 to partition do
        write(' ',A[r,c]:15);
      writeln;
    end;
end prtmat ;

procedure makmat(var A: matrix; var b: rvector);
  var
    r,c,k: idxrng;
    t,rnd: real;
begin                                   make nonsingular matrix and
  for r:=1 to rank do                    and corresponding right hand
    begin t := 0.0;                      side having a solution where
      for c := 1 to rank do              every element is equal to 1.0
        begin
          if c<=r then
            begin rnd:=uniform; t:=t+rnd;
              A[r,c] := rnd
            end
          else A[r,c] := 0.0
        end;
      b[r] := t;
    end;
  for r:=1 to rank-1 do                 perturbate rows
    begin
      if random(0) < 0.5 then
        begin t := uniform;
          b[r] := b[r] + t*b[rank];
          for c := 1 to rank do
            A[r,c] := A[r,c] + t* A[rank,c];
        end;
    end;
  for c:=1 to rank do                   interchange arbitrary columns
    if random(0) < 0.5 then
      for r := 1 to rank do
        begin
          t := A[r,c];
          A[r,c] := A[r,1];
          A[r,1] := t;
        end;
end makmat ;

{*****************************************************************************}
  procedure DAXPY adds a row vector to the product of a constant times
  another row vector

  Thomas R. Turner, IBM Rochester
  transcribed from LINPACK version dated 03/11/78
  Jack Dongarra
{*****************************************************************************}

procedure DAXPY(var r,pivotrow: rvector; multiplier: real; j: idxrng);
  var c: idxrng;
begin
  for c:=j to rank do
```

```
    r[c] := r[c] + multiplier*pivotrow[c]
end DAXPY ;

 ***********************************************************************************************
  procedure DSCAL scales a column vector by a constant multiplier

  Thomas R. Turner, IBM Rochester
  transcribed from LINPACK version dated 03/11/78
  Jack Dongarra
 ***********************************************************************************************

procedure DSCAL(var A: matrix; i,c: idxrng; multiplier: real);
  var r: idxrng;
begin
  for r:=i to rank do
    A[r,c] := multiplier*A[r,c]
end DSCAL ;

 ***********************************************************************************************
  function IDAMAX finds the index of the element having the largest
  magnitude

  Thomas R. Turner, IBM Rochester
  transcribed from LINPACK version dated 03/11/78
  Jack Dongarra
 ***********************************************************************************************

function IDAMAX (var A: matrix; i,c: idxrng): idxrng;
  var t,max: real; pivotrow,r: idxrng;
begin
  max := abs(A[i,c]);                        assume row i has largest
  pivotrow := i;                              element in column c
  for r:=i+1 to rank do                      loop to find largest element
    begin t:=abs(A[r,c]);                     and corresponding index
      if max < t then
        begin max := t; pivotrow := r end;
    end;
  IDAMAX := pivotrow;
end IDAMAX ;


 ***********************************************************************************************
  procedure DGEFA factors a real matrix by gaussian elimination.

  Thomas R. Turner, IBM Rochester
  transcribed from LINPACK version dated 08/14/78
  Cleve Moler, University of New Mexico
 ***********************************************************************************************

procedure DGEFA (var A     : matrix          input  - matrix to be factored
                                             output - an uppeer triangular
                                              matrix and the multipliers
                                              which were used to obtain it
                                              the factorization can be
                                              written A = L*U where L is a
                                              product of permutation and
                                              unit lower  triangular mat-
                                              rices and U is upper triangu-
                                              lar.
```

```
                ;    lda   : idxrng          leading dimension of A
                                              not used in pascal
                ;    n     : idxrng          input  - the order of matrix A
                ;var ipvt  : ivector         output - an integer vector of
                                              pivot indices. the row inter-
                                              change record
                ;var info  : integer         output -
                                              = 0 normal value
                                              = k if U[k,k] = 0.0. this is
                                                  not an error condition
                                                  for this procedure, but
                                                  it does indicate DGESL
                                                  will divide by zero if
                                                  called.
                );
  var
    multiplier,t                : real;
    r,c,k,pivotrow              : idxrng;
begin DGEFA
 ********************************************************************************
  gaussian elimination with partial pivoting
 ********************************************************************************
  info := 0;                                  assume all diagonal elements
                                               will be non zero
  for c:=1 to rank-1 do
    begin r := c;                             initialize row index
      pivotrow := IDAMAX(A,r,c);              find pivot index
      ipvt[r] := pivotrow;                    record pivot index
      if abs(A[pivotrow,c]) < smallreal then
        info := c                              zero pivot implies column c
      else                                      is already triangularized
        begin
          if pivotrow <> r then               interchange rows if pivotrow
            for k:=c to rank do                is not current row "r"
              begin
                t:=A[r,k];
                A[r,k]:=A[pivotrow,k];
                A[pivotrow,k]:=t
              end;
          pivotrow:=r;                       interchange pivot row and cur-
                                              rent row indices
          multiplier:=-1/A[pivotrow,c];      compute multipliers
          DSCAL(A,pivotrow+1,c,multiplier);

          for r:=pivotrow+1 to rank do       row elimination
            DAXPY(A[r],A[pivotrow],A[r,c],c+1);
        end if A[pivotrow,c] ;
    end c ;
  if abs(A[n,n]) < smallreal then info := n;
  ipvt[rank] := rank;
end DGEFA ;


 ********************************************************************************
  procedure DGESL solves the real system ax = b using the factors
  computed by DGEFA

  error condition
  a division by zero will occur if the input factor contains a zero on
```

```
    the diagonal. technically this indicates singularity but it is often
    caused by improper arguments. it will not occur if the subroutines are
    called correctly and if DGEFA has set info to zero.

    Thomas R. Turner, IBM Rochester
    transcribed from LINPACK version dated 08/14/78
    Cleve Moler, University of New Mexico, Argonne National Lab.
 ********************************************************************************

  procedure DGESL (var A    : matrix          input  - factored matrix
                                               output from DGEFA
                   ;    lda  : idxrng         leading dimension of a
                                               not used in this version
                   ;    n    : idxrng         input  - the order of matrix A
                   ;var ipvt : ivector        input  - an integer vector of
                                               pivot indices. the row inter-
                                               change record
                   ;var b    : rvector        input  - the right hand side
                                               column vector
                                              output - the solution vector x
                   ;    job  : integer        input  -
                                               = 0 to solve Ax = b
                                               = k to solve trans(A)x = b
                                               not used in this version
                                                   job must be zero, as this
                                                   implementation does not
                                                   solve trans(A)x = b
                  );
    var
      t                          : real;
      r,c,i,pivotrow             : idxrng;

  begin DGESL
    for r := 1 to rank do                      solve Ly = b
      begin
        pivotrow := ipvt[r];
        if r<>pivotrow then
          begin
            t := b[pivotrow];
            b[pivotrow] := b[r];
            b[r] := t;
          end;
        for c:=r+1 to rank do
          b[c] := b[c] + A[c,r]*b[r];
      end;
    for r := rank downto 1 do                  solve Ux=y
      begin
        t := b[r];
        for c := r+1 to rank do                backward substitution
          t := t - A[r,c]*b[c];
        b[r] := t/A[r,r];
      end;
  end DGESL ;

begin lintest
  makmat(A,b);
  prtmat(A);
  prtrvec(b);
  DGEFAtime := clock;
```

```
  DGEFA(A,rank,rank,ipvt,info);
  DGEFAtime := clock - DGEFAtime;
  writeln('DGEFA time = ',DGEFAtime,' us');
  write('LU form of'); writeln(' matrix A');
  prtmat(A);
  writeln('pivot vector');
  prtivec(ipvt);
  if info = 0 then
    begin
      DGESLtime := clock;
      DGESL(A,rank,rank,ipvt,b,job);
      DGESLtime := clock - DGESLtime;
      write('solution'); writeln(' vector x');
      prtrvec(b);
      writeln('DGESL time = ',DGESLtime,' us');
      mflops := 2.0/3.0*rank*sqr(rank) + 2.0*sqr(rank);
      mflops := mflops/(DGEFAtime + DGESLtime);
      writeln('MFLOPS = ',mflops);
    end;
end lintest .
```

10. APPENDIX B: VECTOR PASCAL VERSION OF LINPACK

```
program fas150;
  const
    partition=4;
    rank=50;
    smallreal = 1.0e-300;
  type
    idxrng  = 1..rank;
    ivector = array[idxrng] of integer;
    rvector = array[idxrng] of real;
    matrix  = array[idxrng] of rvector;
    rvec    = array of real;
  var
    A: matrix;
    b: rvector;
    ipvt: ivector;
    info: integer;
    job: integer;
    DGEFAtime,DGESLtime : integer;
    mflops: real;

  function uniform: real;
    const
      ulo  =  2.0;
      uhi  = 13.0;
   begin
     uniform := round((uhi-ulo)*random(0) + ulo);
   end uniform ;

  procedure prttitle;
    var c: idxrng;
  begin
    write('  ');
    for c:=1 to partition do
      write(' ','-------',c:1,'-------');
    writeln
  end prttitle ;

  procedure prtivec(var v: ivector);
    var c: idxrng;
  begin
    prttitle;
    write('  ');
    for c:=1 to partition do
      write(' ',v[c]:15);
    writeln;
  end prtivec ;

  procedure prtrvec(var v: rvector);
    var c: idxrng;
  begin
    prttitle;
    write('  ');
    for c:=1 to partition do
      write(' ',v[c]:15);
    writeln;
  end prtrvec ;

  procedure prtmat(var A: matrix);
    var r,c: idxrng;
```

```
begin
  prttitle;
  for r:=1 to partition do
    begin
      write(' ',r:1);
      for c:=1 to partition do
        write(' ',A[r,c]:15);
      writeln;
    end;
end prtmat ;

procedure makmat(var A: matrix; var b: rvector);
  var
    r,c,k: idxrng;
    t,rnd: real;
begin                                   make nonsingular matrix and
  for r:=1 to rank do                    and corresponding right hand
    begin t := 0.0;                      side having a solution where
      for c := 1 to rank do              every element is equal to 1.0
        begin
          if c<=r then
            begin rnd:=uniform; t:=t+rnd;
              A[r,c] := rnd
            end
          else A[r,c] := 0.0
        end;
      b[r] := t;
    end;
  for r:=1 to rank-1 do                 perturbate rows
    begin
      if random(0) < 0.5 then
        begin t := uniform;
          b[r] := b[r] + t*b[rank];
          for c := 1 to rank do
            A[r,c] := A[r,c] + t* A[rank,c];
        end;
    end;
  for c:=1 to rank do                   interchange arbitrary columns
    if random(0) < 0.5 then
      for r := 1 to rank do
        begin
          t := A[r,c];
          A[r,c] := A[r,1];
          A[r,1] := t;
        end;
end makmat ;


 ***********************************************************************************************
  function IMAX returns the index of vector element having the largest
  magnitude

  Thomas R. Turner, IBM Rochester
  transcribed from LINPACK version dated 08/14/78
  Cleve Moler, University of New Mexico
 ***********************************************************************************************
function IMAX(v: rvec):idxrng;
  var r,l,maxi: idxrng; max,t: real;
begin l := low(v);
```

```
  max := abs(v[1]); maxi := 1;
  for r := 1+1 to high(v) do
    begin t := abs(v[r]);
      if t > max then
        begin max := t; maxi := r end
    end;
  IMAX := maxi
end IMAX ;

 ***********************************************************************************************
  procedure DGEFA factors a real matrix by gaussian elimination.

  Thomas R. Turner, IBM Rochester
  transcribed from LINPACK version dated 08/14/78
  Cleve Moler, University of New Mexico
 ***********************************************************************************************

procedure DGEFA (var A    : matrix          input  - matrix to be factored
                                            output - an uppeer triangular
                                             matrix and the multipliers
                                             which were used to obtain it
                                             the factorization can be
                                             written A = L*U where L is a
                                             product of permutation and
                                             unit lower  triangular mat-
                                             rices and U is upper triangu-
                                             lar.
                 ;    lda  : idxrng        leading dimension of A
                                             not used in pascal
                 ;    n    : idxrng        input  - the order of matrix A
                 ;var ipvt : ivector       output - an integer vector of
                                             pivot indices. the row inter-
                                             change record
                 ;var info : integer       output -
                                             = 0 normal value
                                             = k if U[k,k] = 0.0. this is
                                                 not an error condition
                                                 for this procedure, but
                                                 it does indicate DGESL
                                                 will divide by zero if
                                                 called.
                 );
  var
    multiplier                 : real;
    r,c,pivotrow               : idxrng;
begin DGEFA
 ***********************************************************************************************
  gaussian elimination with partial pivoting
 ***********************************************************************************************
  info := 0;                                 assume all diagonal elements
                                              will be non zero
  for c:=1 to rank-1 do
    begin r := c;                            initialize row index
      pivotrow := IMAX(A[r..rank,c]);        find pivot index
      ipvt[r] := pivotrow;                   record pivot index
      if abs(A[pivotrow,c]) < smallreal then
        info := c                             zero pivot implies column c
      else                                     is already triangularized
        begin
```

```
          if pivotrow <> r then            interchange rows if pivotrow
                                            is not current row "r"
            swap(A[r,c..rank],A[pivotrow,c..rank]);
          pivotrow:=r;                     interchange pivot row and cur-
                                            rent row indices
          multiplier:=-1.0/A[pivotrow,c];  compute multipliers
          A[pivotrow+1..rank,c] := multiplier*A[pivotrow+1..rank,c];

          for r:=pivotrow+1 to rank do     row elimination
            A[r,c+1..rank] := A[r,c+1..rank] + A[r,c]*A[pivotrow,c+1..rank];
        end if A[pivotrow,c] ;
    end c ;
  if abs(A[n,n]) < smallreal then info := n;
  ipvt[rank] := rank;
end DGEFA ;

procedure DGESL (var A    : matrix          input  - factored matrix
                                             output from DGEFA
                 ;    lda  : idxrng          leading dimension of a
                                             not used in this version
                 ;    n    : idxrng          input  - the order of matrix A
                 ;var ipvt : ivector         input  - an integer vector of
                                             pivot indices. the row inter-
                                             change record
                 ;var b    : rvector         input  - the right hand side
                                             column vector
                                            output - the solution vector x
                 ;    job  : integer         input  -
                                             = 0 to solve Ax = b
                                             = k to solve trans(A)x = b
                                             not used in this version
                                                 job must be zero, as this
                                                 implementation does not
                                                 solve trans(A)x = b.
                 );
  var
    t                           : real;
    r,c,i,pivotrow              : idxrng;

  begin DGESL
    for r := 1 to rank-1 do                  solve Ly = b
      begin
        pivotrow := ipvt[r];
        if r<>pivotrow then
          begin
            t := b[pivotrow];
            b[pivotrow] := b[r];
            b[r] := t;
          end;
        c:=r+1;
        b[c..rank] := b[c..rank] + A[c..rank,r]*b[r];
      end;

    b[rank] := b[rank]/A[rank,rank];
    for r := rank-1 downto 1 do              solve Ux=y
      b[r] := (b[r] - A[r,r+1..rank]*b[r+1..rank])/A[r,r];
  end DGESL ;

begin lintest
```

```
  makmat(A,b);
  prtmat(A);
  prtrvec(b);
  DGEFAtime := clock;
  DGEFA(A,rank,rank,ipvt,info);
  DGEFAtime := clock - DGEFAtime;
  writeln('DGEFA time = ',DGEFAtime,' us');
  write('LU form of'); writeln(' matrix A');
  prtmat(A);
  writeln('pivot vector');
  prtivec(ipvt);
  if info = 0 then
    begin
      DGESLtime := clock;
      DGESL(A,rank,rank,ipvt,b,job);
      DGESLtime := clock - DGESLtime;
      write('solution'); writeln(' vector x');
      prtrvec(b);
      writeln('DGESL time = ',DGESLtime,' us');
      mflops := 2.0/3.0*rank*sqr(rank) + 2.0*sqr(rank);
      mflops := mflops/(DGEFAtime + DGESLtime);
      writeln('Rank of Matrix = ',rank:3,' MFLOPS = ',mflops);
    end;
end fas150 .
```