VAX 6000 Series
Vector Processor Programmer's Guide

Order Number: EK–60VAA–PG–001

This manual is intended for system and application programmers writing programs for the VAX 6000 system with a vector processor.

Digital Equipment Corporation

First Printing, June 1990

Printed in U.S.A.

The Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

This document was prepared with VAX DOCUMENT, Version 1.2.

Contents

FIGURES

TABLES

Preface

Intended Audience

This manual is for the system or application programmer of a VAX 6000 system with a vector processor.

Document Structure

This manual has three chapters and an appendix, as follows:

- **Chapter 1, Vector Processing Concepts,** describes basic vector concepts and how vector processing differs from scalar processing.
- **Chapter 2, VAX 6000 Series Vector Processor,** gives an overview of the vector coprocessor and related vector features.
- **Chapter 3, Optimizing with MACRO–32,** using MACRO–32 and FORTRAN programming examples, illustrates particular programming techniques that take advantage of the high performance of the VAX 6000 series vector processor.
- **Appendix A, Algorithm Optimization Examples,** provides examples of optimization in two application areas: equation solvers and signal processing.
- A **Glossary** and **Index** provide additional reference support.

VAX 6000 Series Documents

Documents in the VAX 6000 series documentation set include:

Title	Order Number
VAX 6000–400 Installation Guide	EK–640EA–IN
VAX 6000–400 Owner's Manual	EK–640EA–OM
VAX 6000–400 Mini-Reference	EK–640EA–HR
VAX 6000–400 System Technical User's Guide	EK–640EB–TM
VAX 6000–400 Options and Maintenance	EK–640EB–MG
VAX 6000 Series Upgrade Manual	EK–600EB–UP
VAX 6000 Series Vector Processor Owner's Manual	EK–60VAA–OM
VAX 6000 Series Vector Processor Programmer's Guide	EK–60VAA–PG

Associated Documents

Other documents that you may find useful include:

Title	Order Number
CIBCA User Guide	EK–CIBCA–UG
DEBNI Installation Guide	EK–DEBNI–IN
Guide to Maintaining a VMS System	AA–LA34A–TE
Guide to Setting Up a VMS System	AA–LA25A–TE
HSC Installation Manual	EK–HSCMN–IN
H4000 DIGITAL Ethernet Transceiver Installation Manual	EK–H4000–IN
H7231 Battery Backup Unit User's Guide	EK–H7231–UG
Installing and Using the VT320 Video Terminal	EK–VT320–UG
Introduction to VMS System Management	AA–LA24A–TE
KDB50 Disk Controller User's Guide	EK–KDB50–UG
RA90 Disk Drive User Guide	EK–ORA90–UG
RV20 Optical Disk Owner's Manual	EK–ORV20–OM

Title	Order Number
SC008 Star Coupler User's Guide	EK–SC008–UG
TK70 Streaming Tape Drive Owner's Manual	EK–OTK70–OM
TU81/TA81 and TU81 PLUS Subsystem User's Guide	EK–TUA81–UG
ULTRIX–32 Guide to System Exercisers	AA–KS95B–TE
VAX Architecture Reference Manual	EY–3459E–DP
VAX FORTRAN Performance Guide	AA–PB75A–TE
VAX Systems Hardware Handbook — VAXBI Systems	EB–31692–46
VAX Vector Processing Handbook	EC–H0419–46
VAXBI Expander Cabinet Installation Guide	EK–VBIEA–IN
VAXBI Options Handbook	EB–32255–46
Vector Processing Concepts Course	EY–9876E–SG
VMS Installation and Operations: VAX 6000 Series	AA–LB36B–TE
VMS Networking Manual	AA–LA48A–TE
VMS System Manager's Manual	AA–LA00A–TE
VMS VAXcluster Manual	AA–LA27A–TE
VMS Version 5.4 New and Changed Features Manual	AA–MG29C–TE

1 Vector Processing Concepts

This chapter presents a brief overview of vector processing concepts. Sections include:

- Scalar vs. Vector Processing
- Types of Vector Processors
- Vectorizing Compilers
- Vector Registers
- Pipelining
- Stripmining
- Stride
- Gather and Scatter Instructions
- Combining Vector Operations to Improve Efficiency
- Performance

1.1 SCALAR VS. VECTOR PROCESSING

Vector processing is a way to increase computer performance over that of a general-purpose computer for certain scientific applications. These include image processing, weather forecasting, and other applications that involve repeated operations on groups, or arrays, of elements. A vector processor is a computer optimized to execute the same instruction repeatedly. For example, consider the process of adding 50 to a set of 100 numbers. The advantage of a vector processor is its ability to perform this operation with a single instruction, thus saving significant processing time.

In computer processors, a vector is a list of numbers, a set of data, or an array. A scalar is any single data item, having one value. A scalar processor is a traditional central processing unit (CPU) that performs operations on scalar numbers in sequential steps. These types of computers are known as single-instruction/single-data (SISD) computers because a single instruction can process only one data item at a time.

A list of elements can be placed in an array. The array is defined by giving each element and its location. Example: a_{12} is the value *a* located at row 1 and column 2. The dimensions of the array are *m* and *n*. An array element is a single value in an array, such as a_{12} below.

$$\begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{vmatrix}$$

A one-dimensional array consists of all elements in a single row or single column. A one-dimensional array can be expressed as a single capital letter such as A, B, or C. Collectively, the elements within a vector are noted by A(I), B(I), C(I), and so forth. Example: B and C are vectors, where:

```
B = (-3, 0, 2)
```

$$C = \begin{bmatrix} 9 \\ -5 \end{bmatrix}$$

The elements B(I) are –3, 0, and 2. The elements C(I) are 9 and –5.

1.1.1 Vector Processor Defined

A vector processor is a computer that operates on an entire vector with a single vector instruction. These types of computers are known as single-instruction/multiple-data (SIMD) computers because a single vector instruction can process a stream of data.

A traditional scalar computer typically operates only on scalar values so it must therefore process vectors sequentially. Since processing by a vector computer involves the concurrent execution of multiple arithmetic or logical operations, vectors can be processed many times faster than with a traditional computer using only scalar instructions.

1.1.2 Vector Operations

A computer with vector processing capabilities does not automatically provide an increase in performance for all applications. The benefits of vector processing depend, to a large degree, on the specific techniques and algorithms of the application, as well as the characteristics of the vector-processing hardware.

Operations can be converted to code to be run on a vector processor if they are identical operations on corresponding elements of data. That is, each operation is independent of the previous step, as follows:

```
A(1) = B(1) + C(1)
A(2) = B(2) + C(2)
A(3) = B(3) + C(3)
   .
   .
   .
A(n) = B(n) + C(n)
```

To create the vector, A(1:*n*), the same function [adding B(*i*) to C(*i*)] is performed on different elements of data, where *i* = 1,2,3 ... *n*. Notice that the equation for A(3) does not depend on A(1) or A(2). Therefore, all these equations could be sent to a vector processor to be solved using one instruction. Two vectors can be added together if both vectors are of the same order; that is, each vector has the same number of elements, *n*. The sum of two vectors is found by adding their corresponding elements. For example, if B = [2, –1, –3] and C = [3, 5, 0], then

```
A =[2+3, -1+5, -3+0] = [5, 4, -3]
```

A scalar processor operates on single quantities of data. Consider the following operation: A(I) = B(I) + 50. As illustrated in Figure 1–1, five separate instructions must be performed, using one instruction per unit of time, for each value from 1 to I_{max} (some CPUs may combine steps and use fewer units of time):

1. Load first element from location B.
2. Add 50 to the first element.
3. Store the result in location A.
4. Increment the counter.
5. Test the counter for index I_{max}.

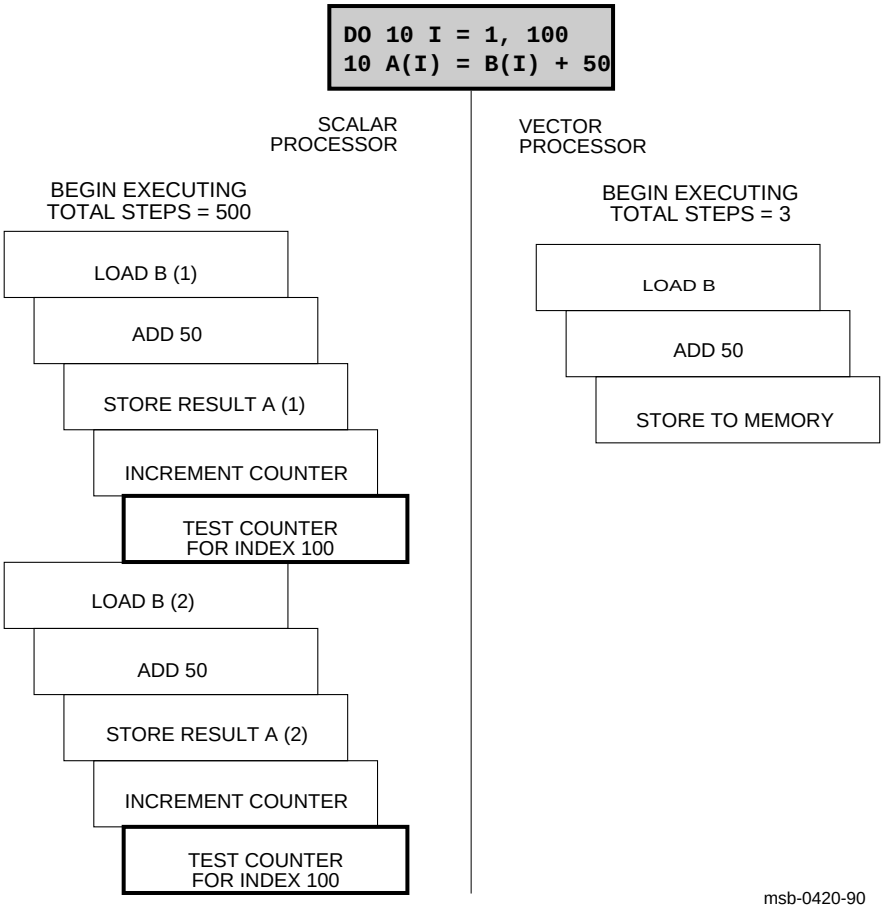
If I_{max} is reached, the operation is complete. If not, steps 1 through 5 are repeated. To calculate A(I) for 100 elements using these instructions (5 X 100), or 500 scalar instructions, takes 500 units of computer time.

Since a vector processor operates on complete vectors of independent data at the same time, only three instructions are needed to perform the same operation A(I) using a vector processor:

1. Load the array from memory into the vector processor register B.
2. Add 50 to all elements in the array, placing the results in register A.
3. Store the entire vector back into memory.

The flow of data optimizes the use of memory and reduces the overhead to perform each operation. Within the vector processor, much the same processing may occur as in the scalar processor, but the vector processor is optimized to do it faster. It is important to remember that vector operations generate the same result as scalar operations.

Figure 1–1 Scalar vs. Vector Processing



1.1.3 Vector Processor Advantages

Vector processors have the following advantages:

- A vector processor can use the full bandwidth of the memory system for loading and storing an array. Unlike the scalar processor, which accepts single values at a time from memory, the vector processor accepts any number up to its limit, say 64 elements, at a time. The vector processor processes all the elements together and returns them to memory.
- The vector processor eliminates the need to check the array index as often. Since all values, up to the vector processor limit, are operated upon at the same time, the vector processor does not have to check the index for each element and each operation.
- The vector processor can free the scalar processor to do further scalar operations. While the vector processor is doing operations other than transferring data to or from memory, the scalar processor can do other functions. This process is in contrast to a scalar processor performing a math operation where the scalar processor must wait until the calculation is complete before proceeding.
- The vector processor runs certain types of applications very fast, since it can be optimized for particular types of calculations.

1.2 TYPES OF VECTOR PROCESSORS

Vector processors are classified according to two basic criteria:

- How closely coupled they are to their scalar coprocessor—whether they are attached or integrated
- How they retrieve vector data—whether they are memory or register processors

1.2.1 Attached vs. Integrated Vector Processors

In general, there are two types of vector processors: attached and integrated. An attached vector processor (also known as an array processor) consists of auxiliary hardware attached to a host system that consists of some number of scalar processors. An attached vector processor, which generally has its own memory and instruction set, can also access data residing in the host main memory. It is typically attached by a standard I/O bus and is treated by a host processor as an

I/O device, controlled under program direction through special registers and operating asynchronously from the host. Program data is moved back and forth between the attached processor and the host with standard I/O operations. The host processor requires no special internal hardware to use an attached vector processor.

There is no "pairing" of a host processor to an attached vector processor. A system can have multiple host scalar processors and one attached vector processor. Some systems can also have one host processor and a number of attached vector processors, all driven by a program executing on the host.

Because it runs in parallel with its host scalar CPU, an attached vector processor can give good performance for the proper applications. However, attached vector processors can be difficult to program, and the need to use I/O operations to transfer program data can result in very high overhead when transferring data between processors. If the data format of the attached processor is different from that of the host system, input and output conversion of the data files will be required.

To perform well on an attached vector processor, an application must have a high percentage of vector operations that need no I/O support from the host. Also, the computational time of those vector operations should be long compared to any required I/O operations.

An integrated vector processor, on the other hand, consists of a coprocessor that is tightly coupled with its host scalar processor; the two processors are considered a pair. The scalar processor is specifically designed to support its vector coprocessor, and the vector processor instruction set is implemented as part of the host's native instruction set. The two processors share the same memory and transfer program instructions and data over a dedicated high-speed internal path. They may also share special CPU resources such as cache or translation buffer entries. Since they share a common memory, no I/O operations are needed to transfer data between them. Thus, programs with a high ratio of data access to arithmetic operations will perform more efficiently on an integrated vector processor than on an attached vector processor.

An integrated vector processor can run synchronously or asynchronously with its scalar coprocessor, depending on the hardware implementation. When the scalar processor fetches and decodes a vector instruction, it passes the instruction to the vector processor. At that point, the scalar processor can either wait for the vector processor to complete the instruction, or it can continue executing and synchronize with the vector processor at a later time. Integrated processors that have this

ability to overlap vector and scalar operations can give better performance than those that do not.

1.2.2 Memory vs. Register Integrated Vector Processors

There are two types of integrated vector processor architectures: memory-to-memory and register-to-register.

In a memory-to-memory architecture, vector data is fetched directly from memory into the function units of the vector processing unit. Once the data is operated on, the results are returned directly to memory.

With a register-to-register (or load/store) architecture, vector data is first loaded from memory into a set of high-speed registers. From there it is moved into the function units and operated on. The resulting data is not returned to the registers until all operations are complete, at which point the vector data is stored back in memory.

For applications that use very long vectors (on the order of thousands of elements), a memory-to-memory architecture works quite well. Once the overhead involved in starting the vector operation is completed, results can be produced at the rate of one element per cycle. On the other hand, with a register-to-register architecture, only a limited segment of the array can be processed at once, and the load/store overhead (or latency) must be paid over and over. With long vectors, this overhead can reduce the performance advantage of high-speed registers.

However, several hardware techniques can be implemented by a register-to-register architecture that can help amortize this load/store overhead. By using techniques such as chaining and instruction overlap, multiple operations can be executed concurrently on the same set of vector data while that data is still in the vector registers. Intermediate (temporary) values need not be returned to memory. Such techniques are not possible with a memory-to-memory architecture.

1.3 VECTORIZING COMPILERS

Developing programs to take maximum advantage of a specific vector processor requires a great deal of knowledge of, and attention to, the particular vector computer hardware. Fortunately most applications that benefit from vector processing can be written in a high-level programming language, such as FORTRAN, and submitted to a vectorizing compiler for that language. The primary function of a vectorizing compiler is to analyze the source program for combinations of arithmetic operations

for which vectorization will yield correct results and generate vector instructions for those operations. If the compiler cannot be certain that a particular expression can be correctly vectorized, it will not vectorize that portion of the source code. The vectorizing compiler can reorganize sections of the program code (usually inside formal loops) that can be vectorized.

Certain portions of all applications are nonvectorizable. Some programming techniques, by their nature, cannot be vectorized. For example, conditional branches into or out of a loop make it impossible for the compiler to know the range of the loop (that is, the vector length) before the code is executed.

In other instances, there may be an unclear relationship between multiple references to the same memory location (called an unknown dependency). In such a relationship, the final value of the location may or may not depend on serial execution of the code, and the compiler does not have enough information to determine whether it can vectorize.

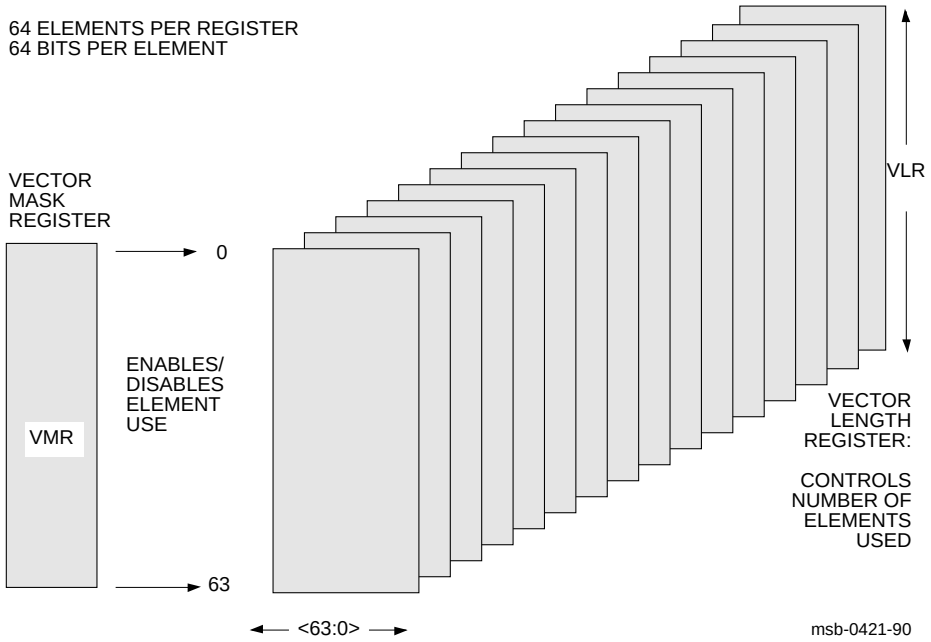
Finally, there may be instances of constructs that could be vectorized but are not. The compiler may not be sophisticated enough to do so, the compiler may determine that vectorization would not be profitable in terms of performance, or the compiler may have insufficient information to determine that vectorization is safe.

1.4 VECTOR REGISTERS

A scalar register is a location in fast memory where data is stored or status bits can be placed to be read at a later time. A register has a set length, say 32 bits, or four consecutive 8-bit bytes.

A vector register is considerably larger. With the VAX, the vector register has a maximum length of 64 elements. Each element can contain up to 64 bits. The elements used can be enabled or disabled by setting bits in a Vector Mask Register (VMR). The programmer usually determines the range, or limits the number of elements used through a Vector Length Register (VLR) (see Figure 1–2). This range can vary, for example, from 0 to 64 elements. Of course, if the vector length = 0, no vector elements will be processed.

Figure 1–2 Vector Registers

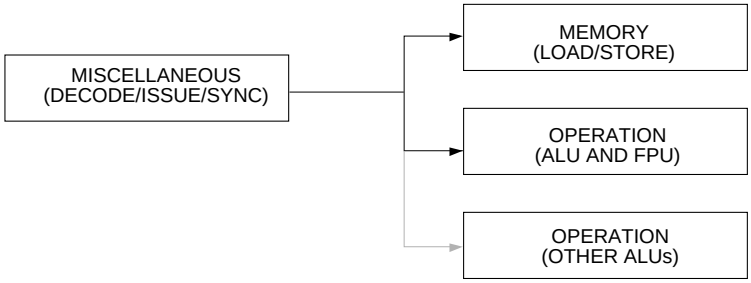


1.5 PIPELINING

A vector function unit, or pipe, performs a specific function within a vector processor and operates independently of other function units. Some vector processors have three function units (see Figure 1–3): one for memory instructions, one for operations (such as add, subtract, and multiply), and one for miscellaneous instructions. Some vector processors have additional function units, specifically to perform additional arithmetic functions.

The performance of the arithmetic and memory function units of a vector processor can be improved using instruction pipelining. Pipelining can be thought of as "assembly line" processing. If a complicated operation can be divided into smaller subprocesses that can then be executed independently by different parts of the function unit, the total time required to execute the operation repeatedly is substantially less than if the operation is performed serially.

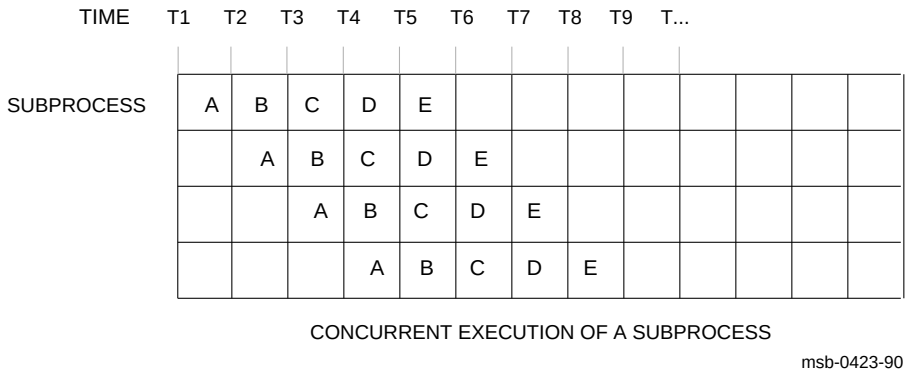
Figure 1–3 Vector Function Units



msb-0422-90

Figure 1–4 shows a concurrent execution of a process divided into four subprocesses. Each subprocess has five operations (A, B, C, D, and E), which start at different times. Operation A might be a load, operation B might be an add, ... and operation E might be a store. There is some overhead in starting the process, but once that overhead (or pipeline latency) is paid, the function unit produces one result per cycle.

Figure 1–4 Pipelining a Process



Because most arithmetic and memory operations can be broken down into a series of one-cycle steps, the function units of a vector processor are generally pipelined. Thus, after initial pipeline latency, the function units can process an entire vector in the number of cycles equal to the length of the input vector—one vector element result per cycle. This time interval (known as a chime) is approximately equal (in cycles) to the length of the vector plus the pipeline latency.

A vector instruction operates on an array of data, so the pipelined execution of vector instructions allows the overlap of multiple iterations of the same vector instruction operating on different data items. The pipeline length equals its number of segments. The maximum number of data elements operated on at any one time equals the pipeline length. Pipelining accommodates the variable array lengths found in vector instructions.

Instruction pipelining can be enhanced by providing multiple parallel pipelines, which operate on different vector elements, within a function unit. As an example, assume a vector has 64 elements. If the vector processor has a function unit with four pipelines, the following processing can be executed in parallel:

```
Pipe 0 operates on elements 0, 4,  8, ... , 60
Pipe 1 operates on elements 1, 5,  9, ... , 61
Pipe 2 operates on elements 2, 6, 10, ... , 62
Pipe 3 operates on elements 3, 7, 11, ... , 63
```

This obviously results in much faster execution than a single pipeline, giving four results per cycle instead of only one. After the pipeline latency, the 64 elements can be processed in 16 cycles rather than in 64.

1.6 STRIPMINING

An array longer than the maximum vector register length of a vector processor must be split into two or more subarrays or vectors, each of which can fit into a vector register. This procedure is known as stripmining (or sectioning), and it is performed automatically by a vectorizing compiler when the source program operates on loops longer than the maximum vector length.

For example, suppose the following FORTRAN loop is vectorized to be run on a vector processor with registers that are 64 elements long:

```
DO 20 I=1,350
  A(I) = B(I) + C(I)
20 CONTINUE
```

Because the vector registers can only hold 64 elements, the compiler vectorizes the loop by splitting the vector into six subvectors to be processed separately.

As typically happens, one subvector in this example is shorter than the full length of a vector register. This short subvector is processed first. Conceptually, the compiler generates vector instructions for the following functions:

```
DO I = 1,30
  A(I) = B(I) + C(I)
ENDDO

DO I = 31,350,64
  DO J = I,I+63
    A(J)=B(J) + C(J)
  ENDDO
ENDDO
```

Note that the compiler must also generate code to set the Vector Length Register to 30 before processing the short vector and then reset it to 64 before processing the remaining vectors.

1.7 STRIDE

To a vector processor, a vector in memory is characterized by a start location, a length, and a stride. Stride is a measure of the number of memory locations between consecutive elements of a vector. A stride equal to the size of one vector element, known as unity stride, means that the vector elements are contiguous in memory. A constant stride greater than the size of one element means that the elements are noncontiguous but are evenly spaced in memory (see Figure 1–5). Most vector processors can load and store vectors that have constant stride.

Not all vector data is constant-strided, however. In some vectors, the distance between consecutive elements in memory is not constant but varies for each pair of elements. Such vectors can be scattered throughout memory and are said to be random-strided or nonstrided (see Figure 1–6). For example, a sparse matrix is generally treated as a nonstrided vector.

Some earlier vector processors did not support this kind of vector. On those systems, special software routines were required to gather the nonstrided vector into a temporary contiguous vector that could be accessed by constant-strided vector memory instructions.

Today, most vector processors support nonstrided vectors with special load and store instructions called gather and scatter.

Figure 1–5 Constant-Strided Vectors in Memory

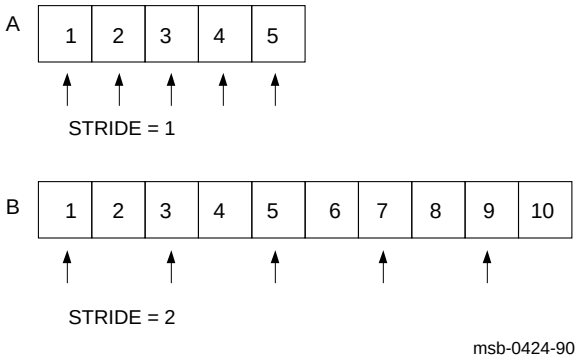
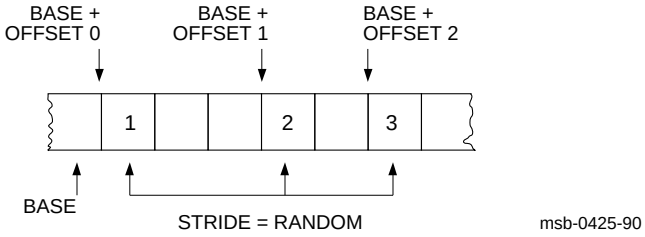


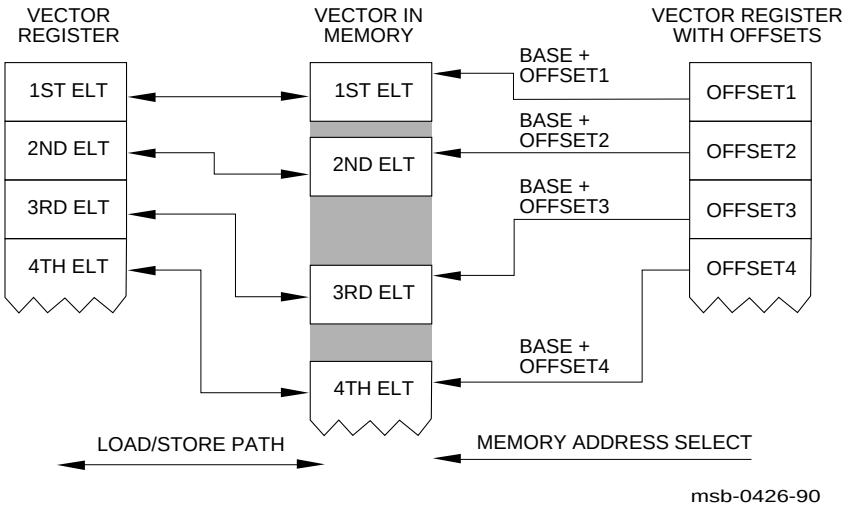
Figure 1–6 Random-Strided Vectors in Memory



1.8 GATHER AND SCATTER INSTRUCTIONS

To support random-strided vectors, gather and scatter instructions operate under control of a vector register that contains an index vector. For each element in the vector, the corresponding element in the index vector contains an offset from the start location of the vector in memory. The gather instruction uses these offsets to "load" the vector elements into the destination register, and the scatter instruction uses them to "store" the vector elements back into memory (see Figure 1–7).

Figure 1–7 Vector Gather and Scatter Instructions



1.9 COMBINING VECTOR OPERATIONS TO IMPROVE EFFICIENCY

Some of the techniques available to increase vector instruction efficiency include overlapping and chaining.

1.9.1 Instruction Overlap

Overlapping instructions involves combining two or more instructions to overlap their execution to save execution time. If a vector processor has independent function units, it can perform different operations on different operands simultaneously. Overlapping provides a significant gain in performance. If a register must be reused or if data is not yet available, overlapping may not be possible.

1.9.2 Chaining

Chaining, a special form of instruction overlap, is possible with multiple function units. Chaining is passing the result of one operation in one function unit to another function unit. For example, an add instruction followed by a store command can "combine" so each element of the vector is stored as soon as the result is obtained. The processor does not have to wait for the add instruction to finish before storing the data.

```
VADD   V1,V2,V3
VSTORE V3
```

As results are generated by the add instruction, they are immediately available for input to the waiting store instruction. The store instruction can then begin processing the data.

Instruction chaining only works if all the data to be processed is available at the beginning of the pipeline. If the result of one operation must be used as input to another operation in the same data stream, instruction chaining cannot be used.

1.10 PERFORMANCE

The performance of scalar computers has been measured for some time using millions of instructions executed per second (MIPS). MIPS is not a good measure of speed for vector processors, since one instruction produces many results. Vector processor execution speed, instead, is measured in millions of floating-point operations per second (MFLOPS). Other abbreviations used are MegaFLOPS, MOPS (millions of operations per second), and RPM (results per microsecond). Some of the largest computers measure speed in gigaFLOPS or billions of floating-point operations per second.

The peak MFLOPS value is a vector processor's best theoretical performance, in terms of the maximum number of floating-point operations per second. For a vector processor having a processor cycle time of 5 nanoseconds and 1 arithmetic unit per pipeline, its peak MFLOPS performance (defined as 1 divided by the cycle time) is determined as follows:

$$(1/5^{-9}) = 0.2X(10^9) = 200MFLOPS$$

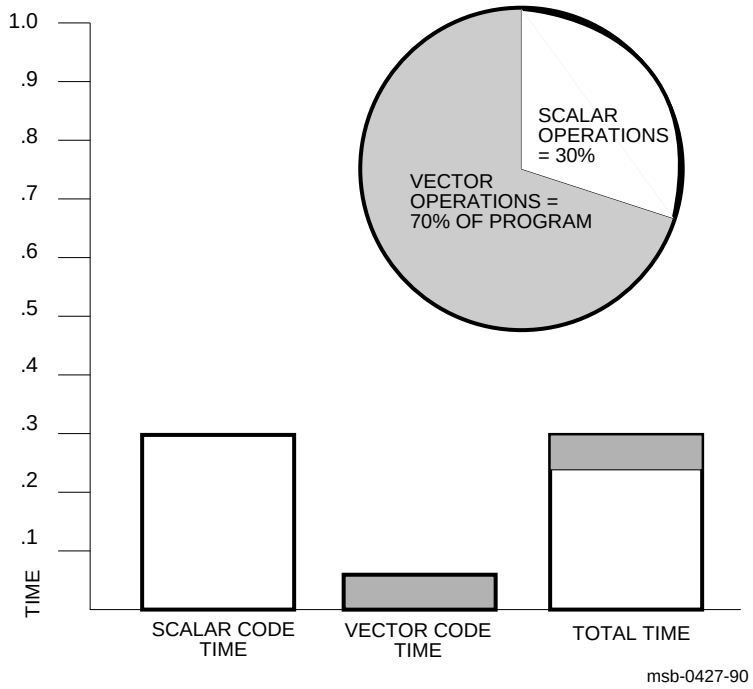
1.10.1 Amdahl's Law

Amdahl's Law indicates that the performance of an application on a computer system with two or more processing rates is dominated by the slowest process. Vector processing is faster than scalar processing executing the same operation, yet the primary factor that determines a computer's speed is the speed of the scalar processor (see Figure 1–8).

Amdahl's law, expressed as an equation, gives the time (T) to perform N operations as:

```
T = N X (%scalar operations X time/scalar operation +
    %vector operations X time/vector operation)
```

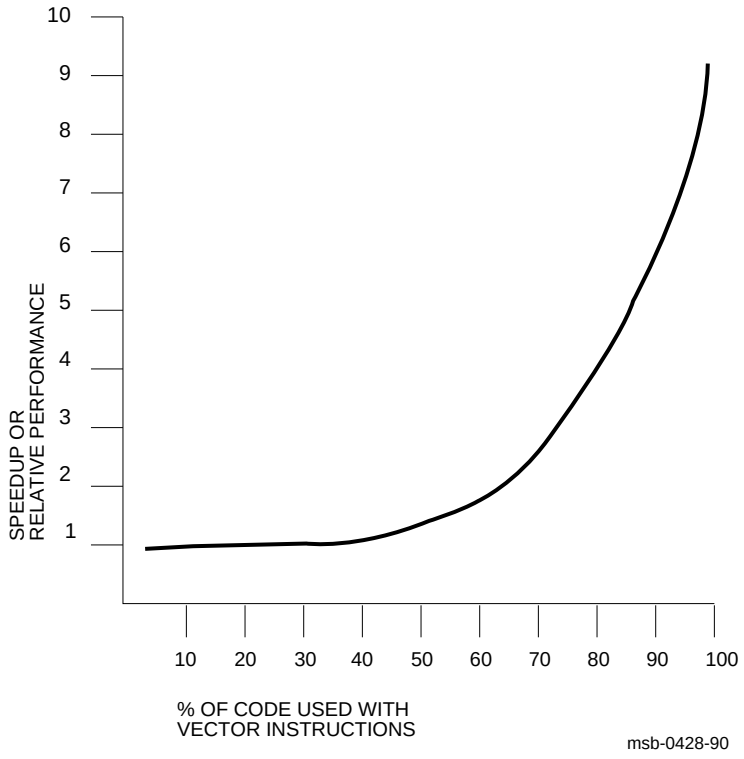
Figure 1–8 Computer Performance Dominated by Slowest Process



1.10.2 Vectorization Factor

Some computer programs use only a portion of the code during the majority of the execution time. For example, a program may spend most of its time doing mathematical calculations, which comprise only 20% of its code. The vectorization factor may be defined as the percentage of the original scalar execution time that may be vectorized. Figure 1–9 shows how the performance increases as more code is vectorized.

Figure 1–9 Computer Performance vs. Vectorized Code



Consider a scalar program that uses 20% of its code 70% of the time. If this 20% portion of the code is converted to vector processing code, the program is considered to have a vectorization factor of 70%. If the time for the scalar operation is set to 1 and the time for a vector operation is 10%, we have:

```
T  =  N * (.30 * 1 + .70 * .1)
T  =  N * .37
```

If performance (*P*), equals operations performed (*N*) per unit time (*T*) then, with $T = N * .37$:

```
P  =  N / T  = N / (N * .37) = 1 / .37 = 2.7
```

The improved performance, shown in Figure 1–9, would be about 2.7 times faster than a scalar processor. Vectorization factors above 70% achieve performance above the same computer using scalar processing. The speedup ratio is defined as the vector performance divided by the scalar performance.

1.10.3 Crossover Point

The crossover point is the vector length or number of elements at which the vector unit exceeds the performance of the scalar unit for a particular instruction or sequence. To achieve a performance improvement on a given vector processor, a vectorized application should have an average vector length that is larger than the crossover point for that processor and the vector operations used.

The smaller the crossover point, the better. A crossover point of 11 means that DO loops below 11 elements are performed faster using a scalar processor than by using a vector processor. This point is a result of the overhead instructions and time required to set up the vector processor, process the data, and return the solution. This point varies from computer to computer.

Vector operations add some startup overhead, putting a limit on the minimum number of elements in an array. For small arrays, the time to process and compile the data is usually longer than doing the same process on a scalar processor.

2 VAX 6000 Series Vector Processor

This chapter describes the vector processor module for the VAX 6000 series. The basic hardware is briefly described and then the hardware components are discussed from the software perspective.

The chapter includes the following sections:

- Overview
- Block Diagram
- Vector Control Unit
- Arithmetic Unit
- Load/Store Unit
- Vector Processor Registers
- Memory Management
- Cache Memory
- Vector Pipelining
- Instruction Execution

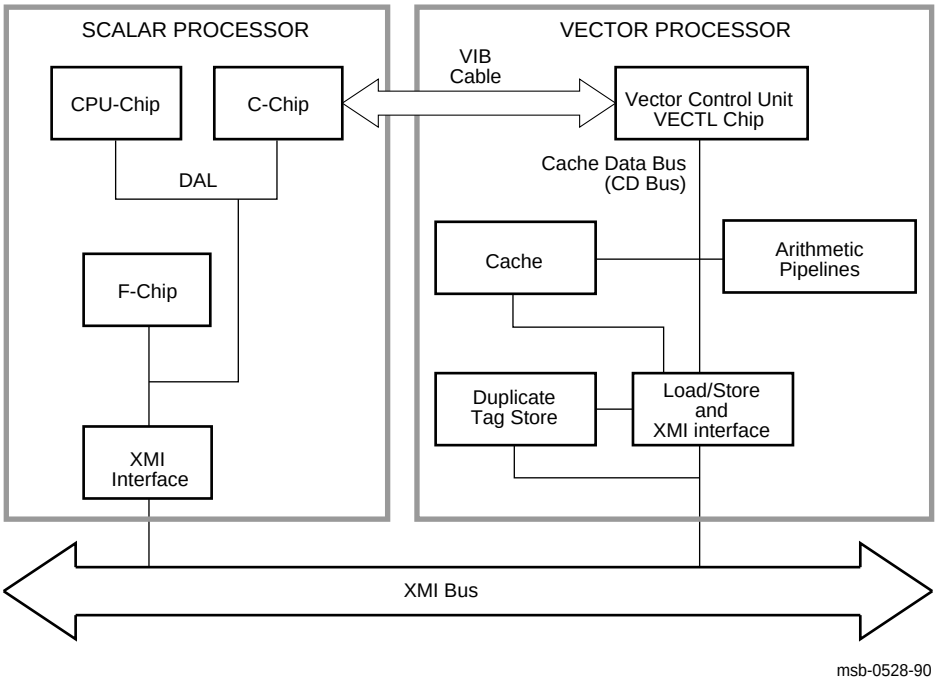
2.1 OVERVIEW

The FV64A vector processor is a single-board option that implements the VAX vector instruction set. This module requires a scalar CPU module for operation. The scalar/vector pair implement the VAX instruction set plus the VAX vector instructions. Figure 2–1 is a block diagram of the scalar/vector pair. The vector processor occupies a slot adjacent to the scalar CPU on the XMI. The two processors are connected by the vector interface bus (VIB) cable.

The C-chip on the scalar module provides the operand and control interface between the scalar CPU and the vector module. This interface is used to issue vector instructions to the vector module, which then executes the instruction, including all memory references necessary to load or store vector registers. The vector processor receives all instructions and returns status to the scalar CPU across the VIB. For memory references, the vector processor has its own independent path to main memory.

The system supports multiple scalar CPUs with a single scalar/vector pair. For a single scalar/vector pair, two memory controllers are required. It also supports a dual scalar/vector pair, for which four memory controllers are required to support the memory traffic.

Figure 2–1 Scalar/Vector Pair Block Diagram



2.2 BLOCK DIAGRAM

The FV64A module is divided into three separate functional units:

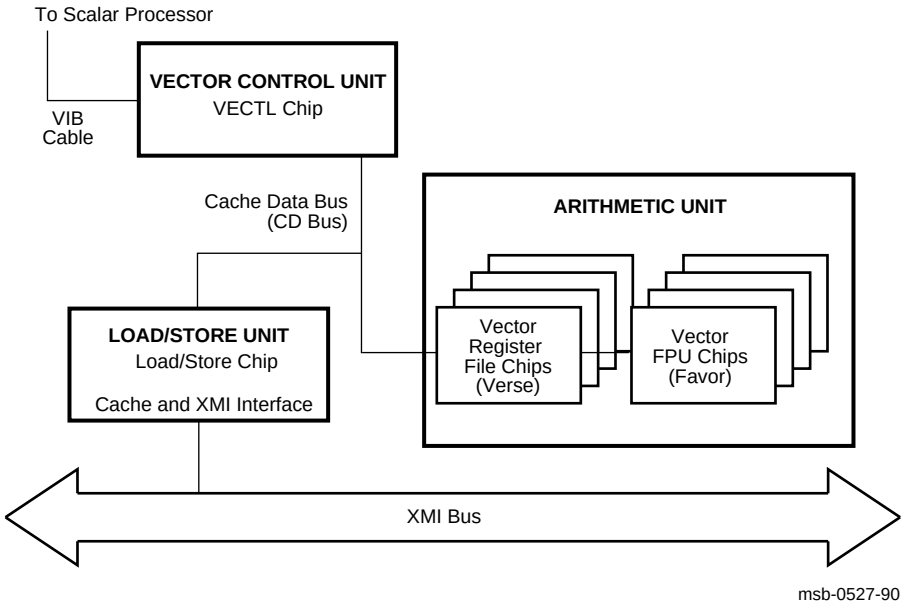
- Vector control unit
- Arithmetic unit
- Load/store unit

All three functional units can operate independently. Figure 2–2 is a block diagram of the vector module.

The FV64A chipset consists of five core chips, as follows:

- Vector instruction issue and scalar/vector interface chip
- Vector register file chip, 4 chips
- Vector arithmetic data path, floating-point unit (FPU) chip, 4 chips
- Load/Store—Vector module translation buffer, cache, and XMI interface controller chip
- Clock generation chip (same as on scalar module)

Figure 2–2 FV64A Vector Processor Block Diagram



2.3 VECTOR CONTROL UNIT

When the vector control unit receives instructions, it buffers the instructions and controls instruction issue to other functional units in the vector module. The vector control unit is responsible for all scalar/vector communication. The vector control unit also contains the necessary register scoreboarding to control instruction overlap. The scoreboard implements the algorithms that permit chaining of arithmetic operations into store operations.

To summarize, the vector control unit performs the following functions:

- Interface to the scalar processor; receives instructions from the scalar module and also returns status.
- Instruction issue. The vector control unit issues instructions to the other functional units of the vector module and maintains a register scoreboard for the detection of interinstruction dependencies.
- Cache data (CD) bus master control. It relinquishes partial control to the load/store unit during execution of load/store instructions.
- Implementation of the Vector Count Register (VCR), Vector Processor Status Register (VPSR), Vector Length Register (VLR), Vector Arithmetic Exception Register (VAER), and Vector Memory Activity Check Register (VMAC).

2.4 ARITHMETIC UNIT

All register-to-register vector instructions are handled by the arithmetic unit. Each vector register file chip contains every fourth element of the vector registers, thus permitting four-way parallelism. These chips receive instructions from the vector contol unit and data from the cache or load/store, read operands from the registers, and write results back into the registers or into the mask register. If two 32-bit operands come over in a single 64-bit transfer, they can be read or written by two separate register file chips.

The register set has four 64-bit ports (one read/write for memory data, two for read operands, and one for writing results). While one instruction is writing its results, a second can start reading its operands, thus hiding the instruction pipeline delay. Variations in pipeline length between instructions are smoothly handled so that no gaps exist in the flow of write data. The register file can hold two outstanding arithmetic instructions in its internal queue. The arithmetic unit executes two arithmetic instructions in about the time it takes one load or store

operation to take place. The data from the register file chip flows to the vector FPU chip.

Input data to the vector FPU chip comes over a 32-bit bus that is driven twice per cycle, and results are returned on a separate 32-bit bus that is driven once per cycle. The two operands for single-precision instructions can be passed in one cycle, while double-precision operands require two cycles. The FPU chip has a throughput of one cycle per single-precision operation, two cycles per double-precision operations, and 10 or 22 cycles per single- or double-precision divide. Its pipeline delay varies for different operations; for example, the pipeline delay is 5 cycles for all longword-type instructions and is 6 cycles for all double-precision instructions except multiply.

2.4.1 Vector Register File Chip

The vector register file chip is the interface between the floating-point processor and the rest of the vector module. Among its features are:

- It contains one quarter of the storage needed to implement the vector registers defined by the VAX vector architecture (2 Kbytes/Verse).
- It provides four ports on the register file: a 64-bit, read/write port to the CD bus for loads and stores, a 32-bit (64-bit internal) read port for operand A, a 32-bit (64-bit internal) read port for operand B, and a 32-bit (64-bit internal) write port for results. A load or store instruction can be writing or reading the registers at one port, and an arithmetic instruction can be reading its operands out of two other ports, and another arithmetic instruction can be writing its results from still another port. All three operations can be done in parallel. When two longword operands are packed into the quadword, two separate vector register file chips can each select the appropriate longword.
- It contains registers for holding two instructions, two scalar operands, the vector length embedded in each instruction, and the vector mask.
- It performs the vector logical and vector merge instructions and formats integer operations so that they can be executed by the FPU.

2.4.2 Vector Floating-Point Unit Chip

The FPU chip is a multi-stage pipelined floating-point processor. Among its features are:

- VAX vector floating-point instructions and data types. The FPU implements instruction and data type support for all VAX vector floating-point instructions as well as the integer multiply operation. Floating-point data types F_, D_, and G_floating are supported.
- High-throughput external interface. The FPU receives two 32-bit operands from the vector register file chip every cycle. It drives back a 32-bit result to the vector register file chip in the same cycle.
- Based on the floating-point accelerator chip (the F-chip) on the scalar module.

2.5 LOAD/STORE UNIT

When a load/store instruction is issued, the load/store unit becomes bus master and controls the internal cache data (CD) bus. Once a load/store instruction starts execution, no further instructions can be issued on the CD bus until it completes. The load/store unit handles the memory reference instructions, the address translation, the cache management, and the memory bus interface.

If a memory instruction uses register offsets, the offset register is first read into a buffer and then each element of the offset register is added to the base. This saves having to turn around the internal bus for each offset read. If a register offset is not used, addresses are generated by adding the stride to the base. This virtual address is then translated to a physical address by using an on-chip 136-entry, fully associative translation buffer (TB). Two entries are checked at once by an address predictor looking for "address translation successful" on the last element. The early prediction permits the scalar processor to be released and appear to be asynchronous on memory reference instructions. The load /store unit handles translation buffer misses on its own but returns the necessary status on invalid or swapped-out pages. Once the scalar processor corrects the situation, the instruction is retried from the beginning.

Once a physical address is obtained, the load/store unit looks it up in the 32K entry tag store. The address is delayed and then passed to the 1-Mbyte cache data store. This delay permits cache lookup to complete before data is written to the cache on store operations. In parallel, the

corresponding register file address is presented to the four register file chips. The data and addresses are automatically aligned for load and store operations to permit the correct reading and writing of the register file and cache data RAMs. Up to four cache misses can be outstanding before the read data for the first miss returns, and hits can be taken under misses. Cache parity errors cause the cache to be disabled, the instruction retried, and when the instruction completes, a soft error interrupt is sent to the scalar processor.

A duplicate copy of the cache tag store is maintained for filtering cache invalidates from the main memory bus. The cache is write through, with a 32-element write buffer, and memory read instructions that hit in the cache can start while the memory write instructions are emptying the write buffer. The cache fill size is 32 bytes. The entire process is pipelined so that a new 64-bit word can be read or written each cycle.

The load/store unit implements the following functions:

- Execution of all load, store, gather, and scatter instructions.
- Virtual address generation logic for memory references.
- Virtual to physical address translation logic, using a translation buffer. A 136-entry TB is part of the load/store unit. The load/store unit also contains the data path and control necessary to implement full VAX memory management (with assistance from the scalar CPU).
- Cache control. The load/store unit supports the tag and data store for a 1-Mbyte write-through data cache. It also supports a duplicate tag store for invalidate filtering.
- XMI interface. The load/store unit serves as the interface between the vector module and the XMI bus. This includes support for four outstanding cache misses on read requests and a 32-entry write buffer to permit half the data from one store/scatter instruction to be held in the buffer. The performance of the high-speed CD bus can thus be isolated from the performance impact of the slower XMI bus.

2.6 VECTOR PROCESSOR REGISTERS

The vector processor has 16 data registers, each containing 64 elements numbered 0 through 63. Each element is 64 bits wide. A vector instruction that reads or writes longwords of F_floating or integer data reads bits <31:0> of each source element and writes bits <31:0> of each destination element.

Other registers used with the data registers are the Vector Length, Vector Count, and Vector Mask Registers (see Figure 2–3). The 7-bit Vector Length Register (VLR) controls how many vector elements are processed. VLR is loaded prior to executing a vector instruction. Once loaded, VLR specifies the length of all subsequent vector instructions until VLR is loaded with a new value.

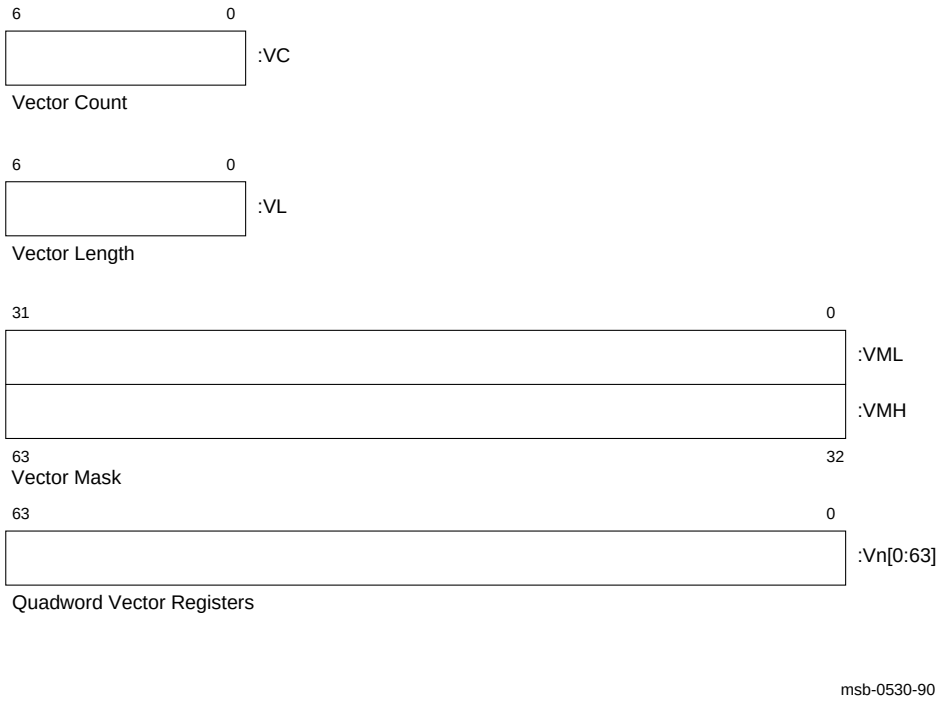
The Vector Mask Register (VMR) has 64 bits, each bit corresponding to an element in a vector register. Bit <0> corresponds to vector element zero. The vector mask is used by the vector compare, merge, IOTA, and all masked instructions.

The 7-bit Vector Count Register (VCR) receives the length of the offset vector generated by the IOTA instruction.

VLR, VCR, and VMR are read and written by Move From/To Vector Processor (MFVP/MTVP) instructions.

The Vector Count and Vector Length Registers are in the vector control unit. The Vector Mask Register and vector data registers are split across the four vector register file chips.

Figure 2–3 Vector Count, Vector Length, Vector Mask, and Vector Registers

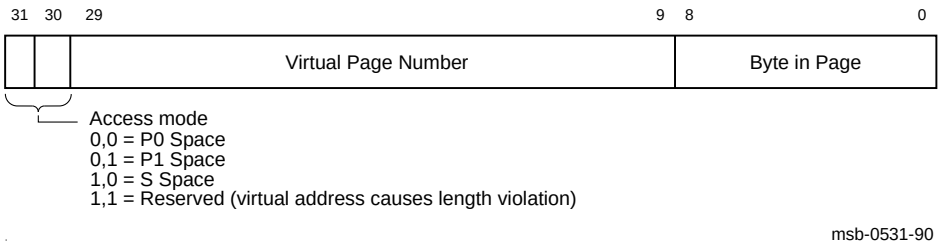


2.7 MEMORY MANAGEMENT

The vector processor implements memory management as described in the *VAX Architecture Reference Manual*.

The 32-bit virtual address is partitioned as shown in Figure 2–4.

Figure 2–4 Virtual Address Format



2.7.1 Translation-Not-Valid Fault

If the V bit = 0 for a page table entry (PTE) which is being used for address translation, and no access violation (ACV) fault has occurred, then the vector module passes status back to the scalar CPU indicating a translation-not-valid (TNV) fault has occurred.

2.7.2 Modify Flows

If the PTE for the page being accessed has V bit = 1, access is a write, no ACV fault has occurred, and the Modify (M) bit is not set, then the memory management unit enters the modify flows. The load/store unit sets the PTE M bit and continues.

2.7.3 Memory Management Fault Priorities

Table 2–1 shows the priority order, from highest to lowest, by which the vector processor reports faults.

Table 2–1 Memory Management Fault Prioritization

ACV	Alignment	TNV	I/O	Modify	Error Reported
1	x	x	x	x	ACV vector, ACV parameter
1	1	x	x	x	ACV vector, align parameter
0	0	1	x	x	TNV vector, TNV parameter
1	0	0	1	x	ACV vector, IOREF parameter
0	0	0	0	1	Execute modify flows
0	0	0	0	0	None; reference OK

2.7.4 Address Space Translation

The memory management hardware translates virtual to physical addresses using the *VAX Architecture Reference Manual* requirements for vector processors.

2.7.5 Translation Buffer

The translation buffer (TB) contains 136 page table entries (PTEs). The TB has 68 associative tags with two PTEs per tag. The TB uses a round-robin replacement algorithm. When a TB miss occurs, two PTEs (one quadword) are fetched from cache. If the fetch from cache results in a cache miss, eight PTEs (one hexword) are loaded into cache from main memory. Two PTEs are installed in the TB.

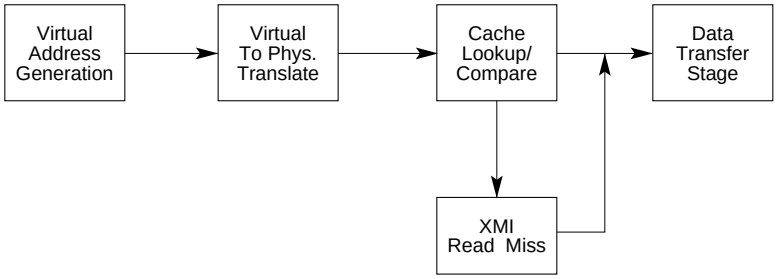
The TB can be invalidated by executing a translation buffer flush. This is accomplished either by writing the VTBIA register or by writing the VTBIS register with the desired virtual address to invalidate a single location.

2.8 CACHE MEMORY

The vector module implements a single-level, direct-mapped cache. In addition, the load/store unit can hold the data and addresses for one complete vector store or scatter instruction. Figure 2–5 shows the flow of address and data in the load/store pipeline.

Each stage is a single or multiple stage based on the 44.44-ns vector module clock. The XMI stage is a multiple of 64 ns, and the time taken depends on the transaction type and the XMI bus activity. All memory references must flow through the cache stage.

Figure 2–5 Address/Data Flow in Load/Store Pipeline



2.8.1 Cache Organization

The vector processor implements a 1-Mbyte cache, direct-mapped, with a fill of a hexword (block) and a hexword allocate (block size). The cache is read allocate, no-write allocate, and write through. There are 32K tags, and each tag maps one hexword block. Each tag contains one block valid bit, a 9-bit tag, and one parity bit. Each data block contains 32 bytes and 8 parity bits, one for each longword.

Associated with each of the 32K main tags is a duplicate tag in the XMI interface. This tag is allocated in parallel with the main tag and is used for determining invalidates. All XMI write command/address cycles are compared with the duplicate tag data to determine if an invalidate should take place. The resulting invalidate is placed in a queue for subsequent processing in the main tag store. Figure 2–6 shows the cache arrangement. Figure 2–7 shows how the physical address is divided.

Figure 2–6 Cache Arrangement

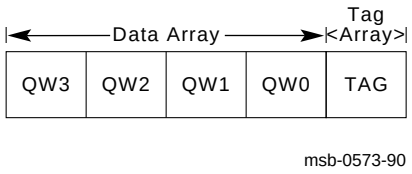
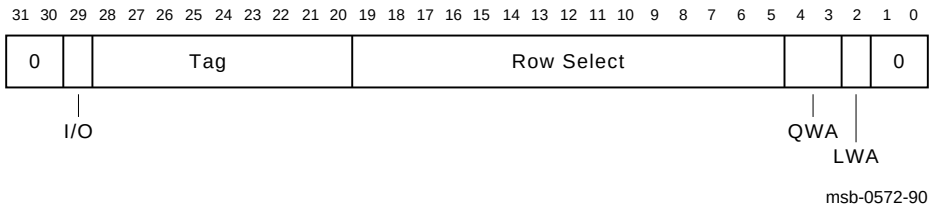


Figure 2–7 Physical Address Division



The physical address passed to the cache is 27 bits long and is longword aligned. Bit <29> is never passed to the cache, because an I/O space reference generates a memory management exception in the translation buffer.Bits <28:20> are compared to the tag field. Bits <19:5> provide the row select for the cache, bits <4:3> supply the quadword address, and bit <2> supplies the longword address.

Figure 2–8 shows how the main tag memory is arranged. The main tag is written with PA<28:20>, and the valid bit covers a hexword block. The parity bit covers the tag and valid bits. The duplicate tag memory is identical to the main tag memory.

Figure 2–9 shows the organization of the cache data. Each cache block contains four quadwords, with eight longword parity bits.

Figure 2–8 Main Tag Memory Organization

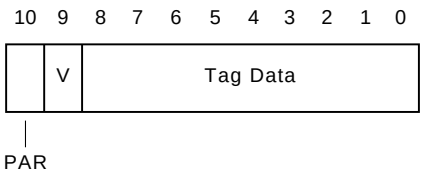
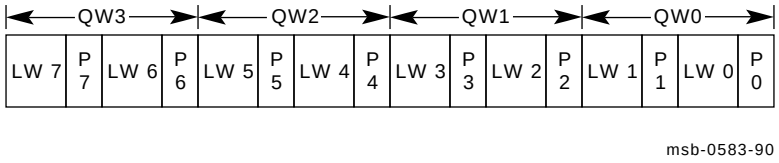


Figure 2–9 Data Cache Logical Organization



2.8.2 Cache Coherency

All data cached by a processor must remain coherent with data in main memory. This means that any write done by a processor or I/O device must displace data cached by all processors.

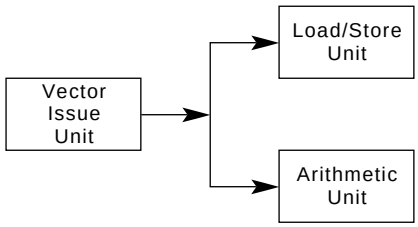
The XMI interface in the load/store unit continuously monitors all XMI write transactions. When a write is detected, the address is compared with the contents of a duplicate tag store to determine if the write should displace data in the main cache. If the write requires that the main cache tag be invalidated, then an invalidate queue entry is generated. The duplicate tag store is a copy of the main tag store. When a main cache tag allocate is performed, the corresponding duplicate tag is also allocated. When an invalidate request is generated, the duplicate tag is immediately invalidated. This mechanism permits full bandwidth operation of the main cache without missing an invalidate request.

The invalidate queue is 16 entries long. In its quiescent state the load /store unit can process invalidates faster than the XMI can generate them. However, during execution of a load or store instruction, the invalidate queue can fill to a level where normal processing must cease, and the invalidate queue is then emptied before an overflow occurs. The number of entries before this mechanism is enabled is nine.

2.9 VECTOR PIPELINING

The vector processor, which is fully pipelined, has three major function units: the vector issue unit, the load/store unit, and the arithmetic unit (see Figure 2–10). These function units operate independently and are fully pipelined. Vector instructions are received by the issue unit from the scalar processor. The issue unit decodes the instruction, performs various checks, and issues the instruction to either the load/store unit or the arithmetic unit. At that point, the issue unit is finished with that instruction and control of the instruction passes to the function unit to which it was issued.

Figure 2–10 Vector Processor Units



msb-0584-90

2.9.1 Vector Issue Unit

The vector issue unit acts as the controller for the vector pipeline. It handles vector instruction decomposition, resource availability checks, and vector instruction issue.

When an instruction is received by the vector module from the scalar processor, the instruction is decomposed into its opcode and operands, and the availability of the requested resources is checked. All source and destination registers must be available before instruction execution begins. The function unit to be used by the instruction during execution must also be available. The instruction is not issued until all required resources are available.

The availability of registers is handled by a method called scoreboarding. The rules governing register availability depend on the type of instruction to be issued.

- For a load instruction, the register to be loaded must not be modified by any currently executing (arithmetic) instruction, and it must not be modified or used as input by any currently deferred (arithmetic) instruction.
- For a store instruction, the register to be stored must not be modified by any currently executing or deferred instruction, but it may be in use as input. The exception is when a chain into store may occur. In this case the store instruction can be issued while the chaining arithmetic instruction is still executing.
- For a scatter or gather instruction, the restrictions for a load or store instruction apply, but also the register containing the offset to be used in the scatter or gather instruction must not be modified by any currently executing or deferred instruction.
- For a load or store under mask instruction, the restrictions for a load or store instruction apply, but also the mask register must not be modified by any currently executing or deferred instruction.
- An arithmetic instruction may be issued as soon as the deferred instruction queue of the arithmetic unit is free. Register checking for these instructions is handled by the arithmetic unit.

In general, there must be no outstanding writes to a needed register from prior instructions, and the destination register of the instruction must not be used by a currently deferred instruction.

Once an instruction is issued, it may take multiple cycles before the result of the calculation is available. Meanwhile, in the next cycle the next instruction can be decoded and, if all its issue conditions are satisfied, it can be issued.

2.9.2 Load/Store Unit

The load/store unit handles all cache and memory access for the vector module. The load/store unit includes a five-segment pipeline that can accept a new instruction every cycle. In general, the load/store pipeline handles a single element request at a time. The exception occurs when a load instruction is acting on single-precision, unity vector stride data. In this special case, consecutive elements are paired and then each pair is handled as a single request by the load/store pipeline.

Once a load or store (or scatter or gather) instruction is issued, no further instructions may be issued until a Memory Management Okay (MMOK) is received. The scalar unit is also stalled until the MMOK is received. Chapter 3 suggests certain coding techniques to minimize the impact of this behavior as much as possible.

2.9.3 Arithmetic Unit

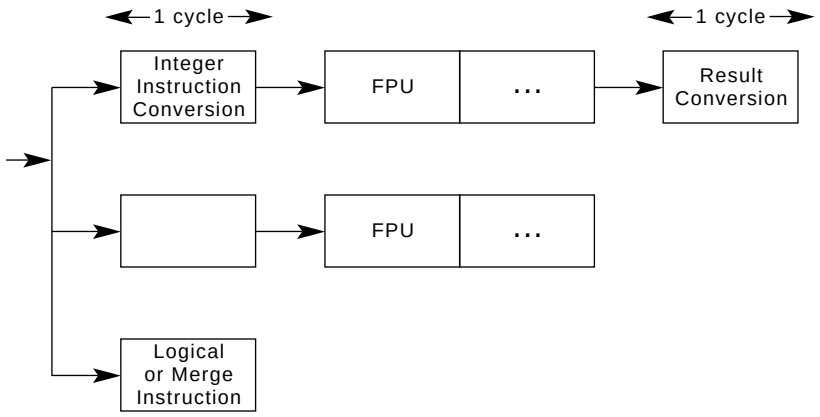
The arithmetic unit is composed of two parts: the vector ALU and the vector FPU. The FPU performs all floating-point instructions as well as compare, shift, and integer multiply. (There is no integer divide vector instruction.) The ALU does everything else, including merge and logical instructions and all initial instruction handling.

The ALU receives instructions from the vector issue unit. One instruction may be queued while another instruction is executing in the arithmetic unit. A queued (or deferred) instruction begins executing as soon as any current instruction completes. Some overlap of instructions is possible if both the current and the deferred instructions require the FPU and are not divide instructions. Also, the second instruction must not begin outputting results before the first instruction completes.

The ALU decodes the instruction and determines the type of operation requested (see Figure 2–11). If the instruction is a Boolean or merge instruction, the ALU performs the required operation. Floating-point instructions, as well as integer, compare, and shift instructions, are sent to the vector FPU for execution.

Once an instruction begins execution in the arithmetic unit, the number of cycles delay (startup time) before the first results are returned depends on the particular instruction executed. With the exception of any type of divide, all instructions return new results each cycle for single-precision data, or every other cycle for double-precision data, following the return of the first results. The total number of cycles required for an instruction to complete depends on the length of the vector and the particular instruction.

Figure 2–11 Vector Arithmetic Unit



An instruction continues executing until all results are completed. A deferred arithmetic instruction begins execution after the instruction in the pipeline completes or when all the following conditions are met:

- The deferred instruction must not be a "short" instruction; that is, the vectors used by the instruction must be at least eight elements in length.
- The current instruction must not be a "long" instruction; that is, the instruction must not require more than two cycles per element to execute. (The divide instructions are the only "long" instructions.)

In other words, overlap of instruction execution can occur if the results of the deferred instruction will not be completed before the last results from the current instruction. The overlap of instructions will be particularly significant for shorter vectors.

All instructions, except floating-point divide instructions, are fully pipelined. For increased performance all arithmetic instructions are executed by four parallel pipelines.

2.10 INSTRUCTION EXECUTION

The vector pipeline is made up of a varying number of segments depending on the type of instruction being executed. Once an instruction is issued, the pipeline is under the control of the load/store unit or the arithmetic unit. The interaction between the different function units of the vector module can greatly affect the performance/execution of vector instructions.

The execution time of a vector instruction can be calculated using the following equation:

```
FC + IC * round_up [ VL / NPP ]
```

where FC is the fixed cost and IC is the incremental cost per vector element, NPP is the number of parallel pipelines, and VL is the length (number of elements) of the vector operand. This can be rewritten in terms of the data as:

```
Startup_latency + Execution_time
```

where Execution_time is a function of vector length.

Note that the execution of D_ and G_floating (64-bit data) type arithmetic instructions (except divide) can only produce results every two cycles due to the bandwidth of the interconnect between the register file and the vector FPU, whereas F_floating type arithmetic instructions (except divide) produce results each cycle.

The execution time of a sequence of instructions is not necessarily equal to the sum of the execution times of the individual instructions. Overlap can occur between arithmetic instructions and load/store instructions as well as between individual arithmetic instructions. It is possible that a sequence of instructions consisting of two arithmetics followed by a load or store can have a total execution time just slightly longer than the execution time of the load or store or equal to the total execution time of the arithmetics, whichever is longer.

In the case of overlap between individual arithmetic instructions, a minimum of one cycle must elapse between the final result of the first instruction and the first result of the following instruction. In other words, when overlap occurs the total execution time decreases. For all overlapping arithmetic instructions, other than the first instruction to enter the empty pipeline, the effective fixed cost (or startup latency) is reduced to a minimum of one cycle.

3 Optimizing with MACRO-32

This chapter discusses optimization features of the VAX 6000 series vector processor. Appendix A provides additional optimization examples. This chapter includes the following sections:

- Vectorization
- Crossover Point
- Scalar/Vector Synchronization
- Instruction Flow
- Overlap of Arithmetic and Load/Store Instructions
- Out-of-Order Instruction Execution
- Chaining
- Cache
- Stride/Translation Buffer Miss
- Register Reuse

3.1 VECTORIZATION

Many loops that VAX FORTRAN decomposes can also be vectorized. VAX FORTRAN performs vectorization automatically whenever /VECTOR is specified at compilation. VAX FORTRAN can vectorize any FORTRAN–77 Standard-conforming program; and vectorized programs can freely call and be called by other programs (vectorized, not vectorized, and non-FORTRAN) as long as both abide by the VAX Calling Standard.

The VAX vector architecture supports most FORTRAN language features, as follows:

- Data types: LOGICAL*4, INTEGER*4, REAL*4, REAL*8, COMPLEX*8, and COMPLEX*16
- Operators: +, -, *, /(floating point), and **
- All VAX FORTRAN intrinsic functions

Although no VAX vector form exists for integer divide operations, VAX FORTRAN vectorizes them by converting them to floating-point operations.

3.1.1 Using Vectorization Alone

Vectorize a program using the following iterative process:

1. Using /CHECK=BOUNDS, compile, debug, and run a scalar version of the program.
2. Compile and run the program using /VECTOR and the suitable vector-related qualifiers.
3. Evaluate execution time and results. The results should be algebraically equivalent to the scalar results; if not, check the DUMMY_ALIASES or ACCURACY_SENSITIVE settings.
   - If performance is adequate, stop.
   - If performance is inadequate, you have similar options as with autodecomposition:
     – Check the /SHOW=LOOPS output to see if CPU-intensive loops vectorized. To vectorize effectively, source code must not contain certain inhibiting constructs such as unknown dependencies. However, you can use LSE diagnostics and add assertions to the source code to overcome dependencies.

- Combine vectorization with decomposition.
- Consider a solution at a higher level hierarchy.
- Retest the program compiling with /VECTOR (or any combination with a parallel qualifier) and return to the start of step 3.

3.1.2 Combining Decomposition with Vectorization

To produce code that executes in parallel and on vector processors, compile /VECTOR with a parallel qualifier. Table 3–1 lists the compilation combinations and their interpretations.

Table 3–1 Qualifier Combinations for Parallel Vector Processing

Combination	Interpretation
/VECTOR/PARALLEL=AUTOMATIC	Performs a dependence analysis on suitable loops and optimizes them for parallel-vector processing; chooses loops and prepares them for vector or parallel processing based on whether they will execute efficiently and produce correct results. In a nested structure, decomposition and vectorization may occur for multiple loops — but no loop is decomposed inside a decomposed loop and no loop is vectorized inside a vectorized loop.
/VECTOR/PARALLEL=MANUAL	Performs a dependence analysis, optimization, and vectorization only; disqualifies from vectorization any loops preceded by the CPAR$ DO_PARALLEL directive from vectorization; in these loops, parses the user-supplied directives.
/VECTOR/PARALLEL	(Same as VECTOR/PARALLEL=MANUAL)
/VECTOR/PARALLEL=(MANUAL,AUTOMATIC)	Same as /VECTOR/PARALLEL=AUTOMATIC except disqualifies loops preceded by CPAR$ DO_PARALLEL. In those loops, only user-supplied directives are parsed. Any loops contained in a manually decomposed loop are disqualified from autodecomposition but not vectorization.

Both parallel and vector processing have certain tradeoff qualities, which affect the aggregate speedup of vector and parallel processing. The combined vector-parallel processing will be somewhat less than the

aggregate speedup of each because of these qualities; however, both CPU time and wall-clock time can be reduced most dramatically when vector and parallel processing are combined.

The qualities involved are as follows:

- Large amounts of vector load-stores can create a bottleneck in the system. On the other hand, small amounts of CPU work can cause the parallel processing startup overhead itself to become a bottleneck. Vector operations have smaller startup overhead than parallel processing, so they amortize this CPU expense much sooner. However, vector processing demands more from memory than parallel processing (on scalar CPUs) because one vector load or store can affect up to 64 elements, whereas a scalar load or store typically affects only one element.
- Vector processing is "free" for the scalar CPUs because it is done on a vector processor; both wall-clock time and scalar CPU time are decreased. On the other hand, parallel processing is not free for the scalar CPUs; it can never decrease CPU time. But it can reduce wall-clock time more dramatically than vector processing.

Vectorization can be effectively combined with decomposition:

1. Compile, debug, and run the program serially and in scalar.
2. Evaluate the algorithm and make suitable changes.
3. Unless your algorithm and system environment are especially suitable for parallel processing or you have already decomposed the program, compile, debug, and run the program using /VECTOR first. This is because vectorization is "free," as stated in this section.
4. Using /VECTOR/PARALLEL=AUTOMATIC, recompile, debug, and run the program.
5. Evaluate performance.
    - If performance is adequate, stop.
    - If performance is inadequate, review the /SHOW=LOOPS output and LSE diagnostics and modify the source code as needed for important loops that neither vectorized nor decomposed (this most probably will include adding assertions to resolve unknown dependencies). Then retest the program. If performance is still not acceptable, consider manually decomposing certain loops and look for other bottlenecks such as I/O or other performance inhibitors.

3.1.3 Algorithms

At times it is necessary to consider the algorithm that is represented by the code to be optimized. Some algorithms are not as well suited to vectorization as others. It may be more effective to change the algorithm used or the way it is implemented rather than trying to optimize the existing code. Increasing the work performed in any single loop iteration and increasing the ratio of arithmetic to load/store instructions are two effective methods to consider when optimizing an algorithm for vectorization. Using unity stride rather than nonunity stride and longer vector lengths are other approaches to consider.

3.2 CROSSOVER POINT

For any given instruction or sequence of instructions, there is a particular vector length where the scalar and vector processing of equivalent operations yield the same performance. This vector length is referred to as the crossover point between scalar and vector processing for the given instruction or instruction sequence and varies depending on the particular instruction or sequence. For vector lengths below the crossover point, scalar operations are faster; above the crossover point vector operations are more efficient. A low crossover point is considered a benefit, since it indicates that it is relatively easy to take advantage of the power of the vector processor.

For any single, isolated vector instruction, the crossover point on the VAX 6000 is quite low, generally about 3 elements. But an instruction is not performed in isolation. Taken in the context of a routine or application, other factors affect the performance of the operations on short vectors, in particular whether the data of the short vector is used in other vector operations as well. In general, on the VAX 6000 vectorizing as much code as possible, including short vector length sections, leads to higher performance through more optimal use of cache. Specifically, once a set of data has been operated on by vector instructions, that data will be in the vector cache. A subsequent scalar operation on any of that same data will require that the data be moved out of the vector cache into the scalar cache. A vector operation would not require this data movement and thus is usually more efficient. Overall, the crossover point on the VAX 6000 is low enough that only for isolated operations on short vectors is scalar processing the faster alternative.

3.3 SCALAR/VECTOR SYNCHRONIZATION

For most cases, it is desirable for a vector processor to operate concurrently with the scalar processor so as to achieve best performance. However, there are cases where the operation of the vector and scalar processors must be synchronized to ensure correct program results. Rather than forcing the vector processor to detect and automatically provide synchronization in these cases, the architecture provides software with special instructions to accomplish the synchronization. These instructions synchronize the following:

- Exception reporting between the vector and scalar processors
- Memory accesses between the scalar and vector processors
- Memory accesses between multiple load/store units of the vector processor

Software must determine when to use these synchronization instructions to ensure correct results.

3.3.1 Scalar/Vector Instruction Synchronization (SYNC)

A mechanism for synchronization between the scalar and vector processors is provided by the SYNC instruction, which is implemented by a Move From Vector Processor (MFVP) instruction. SYNC allows software to ensure that the exceptions of previously issued vector instructions are reported before the scalar processor proceeds with the next instruction. SYNC detects both arithmetic exceptions and asynchronous memory management exceptions and reports these exceptions by taking the appropriate VAX instruction fault. Once it issues the SYNC, the scalar processor executes no further instructions until the SYNC completes or faults.

When SYNC completes, a longword value (which is unpredictable) is returned to the scalar processor. The scalar processor writes the longword value to the scalar destination of the MFVP instruction and then proceeds to execute the next instruction.

When SYNC faults, it is not completed by the vector processor, and the scalar processor does not write a longword value to the scalar destination of the MFVP instruction. Also depending on the exception condition encountered, the SYNC itself takes either a vector processor disabled fault or memory management fault. After the appropriate fault has been

serviced, the SYNC may be returned to through a Return from Exception or Interrupt (REI) instruction.

SYNC only affects the scalar/vector processor pair that executed it. It has no effect on other processors in a multiprocessor system.

3.3.2 Scalar/Vector Memory Synchronization

The scalar processor and the vector processor can access memory at the same time during:

- Asynchronous memory management mode
- Synchronous memory management mode, after the vector processor indicates no memory management exceptions occurred

When the scalar processor and the vector processor access memory at the same time, it may be desirable to synchronize their accesses. Using an MFVP from MSYNC vector control register causes the scalar CPU to stall until previous memory accesses by either the vector processor or the scalar processor are completed and visible to the other. MSYNC is for user software.

Scalar/vector memory synchronization allows software to ensure that the memory activity of the scalar/vector processor pair has ceased and that the resultant memory writes have been made visible to each processor in the pair before the pair's scalar processor proceeds with the next instruction. Two ways are provided to ensure scalar/vector memory synchronization:

- Using MSYNC, which is implemented by the MFVP instruction
- Using the Move From Processor Register (MFPR) instruction to read the Vector Memory Activity Check (VMAC) internal processor register

In the following example, both the vector processor load instruction (VLDL) and the scalar processor move instruction (MOVF) would be using the same BASE memory. MSYNC ensures that the load instruction completes before beginning the move instruction.

```
VLDL BASE, #4, V1
MSYNC R0
MOVF#^F3.0, BASE
```

Scalar/vector memory synchronization does not mean that previously issued vector memory instructions have completed; it only means that the vector and scalar processor are no longer performing memory operations. While both VMAC and MSYNC provide scalar/vector memory synchronization, MSYNC performs significantly more than just that function. In addition, VMAC and MSYNC differ in their exception behavior.

Note that scalar/vector memory synchronization only affects the processor pair that executed it. Other processors in a multiprocessor system are not affected. Scalar/vector memory synchronization does not ensure that the writes made by one scalar/vector pair are visible to any other scalar or vector processor.

Software can make data visible and shared between a scalar/vector pair and other scalar and vector processors by using the mechanisms described in the *VAX Architecture Reference Manual*. Software must first make a memory write by the vector processor visible to its associated scalar processor through scalar/vector memory synchronization synchronization) before making the write visible to other processors. Without performing this scalar/vector synchronization, it is unpredictable whether the vector memory write will be made visible to other processors even by the mechanisms described in the *VAX Architecture Reference Manual*.

Note that waiting for VPSR<BSY> to be clear does not guarantee that a vector write is visible to the scalar processor.

3.3.2.1 Memory Instruction Synchronization (MSYNC)

Once it issues MSYNC, the scalar processor executes no further instructions until MSYNC completes or faults.

When MSYNC completes, a longword value (which is unpredictable) is returned to the scalar processor, which writes it to the scalar destination of the MFVP instruction. The scalar processor then proceeds to execute the next instruction.

Arithmetic and asynchronous memory management exceptions encountered by previous vector instructions can cause MSYNC to fault.

When MSYNC faults, all previously issued scalar and vector memory instructions may not have finished. In this case, the scalar processor writes no longword value to the scalar destination of the MFVP. Depending on the exception encountered by the vector processor, the MSYNC takes a vector processor disabled fault or memory management

fault. After the fault has been serviced, the MSYNC may be returned to through a Return from Exception or Interrupt (REI) instruction.

3.3.2.2 Memory Activity Completion Synchronization (VMAC)

Privileged software needs a way to ensure scalar/vector memory synchronization that will not result in any exceptions being reported. Reading the Vector Memory Activity Check (VMAC) internal processor register with the privileged Move From Processor Register (MFPR) instruction is provided for these situations. It is especially useful for context switching.

Once an MFPR from VMAC is issued by the scalar processor, the scalar processor executes no further instructions until all vector and scalar memory activities have ceased; all resultant memory writes have been made visible to both the scalar and vector processor; and a longword value (which is unpredictable) is returned to the scalar processor. After writing the longword value to the scalar destination of the MFPR, the scalar processor then proceeds to execute the next instruction.

Vector arithmetic and memory management exceptions of previous vector instructions never fault a privileged MFPR from the Vector Memory Activity Check Register and never suspend its execution.

3.3.3 Memory Synchronization Within the Vector Processor (VSYNC)

The vector processor can concurrently execute a number of vector memory instructions through the use of multiple load/store paths to memory. When it is necessary to synchronize the accesses of multiple vector memory instructions, the MSYNC instruction can be used; however, there are cases for which this instruction does more than is needed. If it is known that only synchronization between the memory accesses of vector instructions is required, the Synchronize Vector Memory Access (VSYNC) instruction is more efficient.

If a conflict results within the vector processor for accessing memory, a VSYNC instruction can be used. VSYNC ensures that the current memory access instruction is complete before executing another. This instruction does not affect scalar processor memory access instructions.

VSYNC orders the conflicting memory accesses of vector memory instructions issued after VSYNC with those of vector memory instructions issued before VSYNC. Specifically, VSYNC forces the access of a memory location by any subsequent vector memory instruction to wait for (depend

upon) the completion of all prior conflicting accesses of that location by previous vector memory instructions.

VSYNC does not have any synchronizing effect between scalar and vector memory access instructions. VSYNC also has no synchronizing effect between vector load instructions because multiple load accesses cannot conflict. It also does not ensure that previous vector memory management exceptions are reported to the scalar processor.

3.3.4 Exceptions

There are two categories of exceptions within the vector processor:

- Imprecise exceptions
- Precise exceptions

3.3.4.1 Imprecise Exceptions

Imprecise exceptions can occur within the vector processor when arithmetic instructions are processing. They may be caused by typical arithmetic problems such as division by zero or underflow. Because the vector processor can execute instructions out of order, it is not possible to determine the instruction that caused the exception from the updated program counter (PC). The PC in the scalar processor is pointing further down the instruction stream and cannot be backed up to point at the failing instruction. To report the exception condition in this case, the vector processor disables itself so that the scalar processor will take a vector disable fault when it attempts to dispatch a vector instruction. The vector disable fault handler then determines the cause. When this type of exception occurs, the vector controller sets a bit in the register mask in the Vector Arithmetic Exception Register (VAER) IPR to indicate the destination vector register which received data from the exception. It then informs the scalar CPU of the exception.

When debugging code, it is often necessary to be able to find the precise instruction causing the problem. Inserting a SYNC instruction after each arithmetic instruction will cause the machine to run in precise mode, waiting for each instruction to complete before executing the next. However, it will run much slower than when imprecise exceptions are allowed to occur.

3.3.4.2 Precise Exceptions

The vector processor produces precise exceptions for memory management faults. When a memory management exception occurs, microcode and operating system handlers are used to fix the exception.

The vector processor cannot service Translation Not Valid and Access-Control Violation faults. To handle these exceptions, the vector processor passes sufficient state data back to the scalar CPU. Then if a memory management fault occurs, the microcode can build a vector exception frame on the stack so that vector processor memory manangement exceptions will be handled precisely and the faulting instruction restarted.

To enforce synchronous operation, after a vector load/store operation is issued, the scalar CPU will not issue additional instructions until memory management has completed. To reduce the delay from the issue of a load/store instruction to the issue of the next instruction, the load/store unit has special logic which predicts when load/store instructions can proceed fault free. When the load/store unit knows it can perform all virtual to physical translations without incurring a memory management fault, it issues the MMOK signal to the vector control unit. The scalar CPU is then released to issue more instructions while the load/store unit completes the remainder of the data transfers. This mechanism reduces the overhead associated with providing precise memory management faults.

3.4 INSTRUCTION FLOW

Vector instructions are read from the scalar CPU's I-stream. The scalar issue unit decodes the vector instructions and passes them to the vector CPU. The instructions are decoded by the vector control unit and then issued to the appropriate function unit through the internal bus. Before instruction issue, the instruction is checked against a register scoreboard to verify that it will not use a corrupted register or attempt to modify a register that is already in use. Load, store, scatter, and gather instructions are processed in the Load/Store chip. These instructions either fetch data from memory and place it in the vector register file or write data from the vector register file to memory.

Arithmetic instructions are passed to the arithmetic pipelines by way of control registers in the register file chips. An arithmetic instruction has a fixed startup latency. To minimize the effects of this latency, the arithmetic pipelines support the ability to queue two arithmetic instructions. This permits the arithmetic pipeline controller to start the second instruction without any startup latency. The removal of

startup latency for the second arithmetic instruction (deferred arithemetic instruction) is a benefit in algorithms that require less than eight Bytes /FLOP of load/store bandwidth.

Typical algorithms benefit greatly from the ability to chain an arithmetic operation into a store operation. The vector control unit, along with the ALU unit, implements this capability. The following sections describe by instruction type the flow of instructions in the machine.

3.4.1 Load Instruction

When a load instruction is received by the vector control unit, the destination vector register is checked against outstanding arithmetic instructions. A load instruction cannot begin execution until the register to which it will write is free. A register conflict may occur if the destination register of a load instruction is the same as one of the registers used by a preceding arithmetic instruction. If instruction execution overlap could occur if the load instruction were using a different register, then the register conflict can be eliminated by simply changing the register used.

If there are no register usage conflicts, the instruction is dispatched to the load/store unit. An example of a memory access instruction in assembler notation is as follows:

```
VLDL  base, stride, Vc
```

where:

```
VLD = vector load (load memory data into vector register)
L = longword (Q would equal quadword)
base = beginning of first element
stride = number of memory locations (bytes) between the
         starting address of the first element and the
         next element
Vc = vector register destination result
```

This instruction means:

Load the vector register (Vc) from memory, starting at the base address (base), incrementing consecutive addresses by the stride in bytes. The load operation writes the data from memory into the destination register. The store operation writes the data from the vector register back to memory.

In the load/store instruction, the Vector Length Register (VLR) and the Vector Mask Register (VMR) with the match true/false (T/F) (when the mask operate enable (MOE) bit is set) determine which elements to access in Vc. For longwords, only bits <31:0> may be accessed. The elements can be loaded or stored out of order, because there can be multiple load /store units and multiple paths to memory, a desirable effect of vector processors.

A Modify Intent (MI) bit may be used with the VLD instruction to improve performance for systems that use writeback caches. The MI bit is not used for store or scatter instructions.

During a load operation, the first element in memory at the base address loads into the destination vector register. The next element in memory at the base address plus the stride loads into the next location in the destination vector register. With a vector load/store operation, the stride is constant, so that the third address in memory is the base address plus two times the stride.

3.4.2 Store Instruction

When the vector control unit receives a store instruction, the source vector register is checked against outstanding arithmetic instructions. If there are no conflicts, the instruction is dispatched to the load/store unit. If the source for the store is the destination of the current arithmetic instruction, and the deferred arithmetic instruction does not conflict with the source vector register, and the arithmetic instruction is not a divide, then the vector control unit waits for a signal from the arithmetic unit to indicate that the store operation can start. The instruction is then dispatched to the load/store unit.

During a store operation, the data moves in the opposite direction, from the destination vector register back to memory. The elements of the vector are placed back into memory at the base address plus a multiple of the stride, as shown in the following example:

```
VLDL base,#4,V3     Load vector V3 from memory, starting at
                    the "base" address and obtaining next
                    elements every 4 bytes apart (stride = 4).

VSTL V1,base,#16    Store vector V4 into memory starting at
                    "base" address and placing next
                    elements 16 bytes apart.
```

The data from a store instruction is internally buffered in the chip. This offers the advantage of allowing cache hit load instructions to issue and complete while the write executes over the XMI.

3.4.3 Memory Management Okay (MMOK)

When a memory reference occurs, control is turned over to memory management until an MMOK is returned indicating that all memory references were successful. An algorithm is used to predict when MMOK will be returned, to determine when new instructions can be issued. For every vector element a new last element virtual address is calculated based on the current element virtual address, the number of remaining elements, and the stride. Every element virtual address is compared to the calculated last element virtual address to determine whether both reside in the same two virtual page window. If they do reside within the same two pages and if the current virtual address has been successfully translated, then MMOK is asserted. If not, then the generation of virtual addresses continues.

3.4.4 Gather/Scatter Instructions

An array whose subscript is an integer vector expression is "indirectly addressed." Indirect addressing appearing on the right side of an assignment statement is called a gather operation; on the left side it is known as a scatter, as shown in the following:

```
DO 80 I = 1,95
   J = IPICK(I)
   A(I) = B(J) + C(K(I)+3) * D(I)
80 CONTINUE
```

Array A requires a scatter operation. B and C require gathers.

Loops that contain references to a scattered array or stores into a gathered array [have potential for data dependency, as shown in the following:

```
DO 10 I = 1,N
   A(I) = B(I) + C(I) / D(ID(I))
   B(IB(I)) = X(I) * Y(I)
   D(I) = E(I)**2
   G(JG(I)) = 2. * G(NG(I))
10 CONTINUE
```

Potential data dependency exists for arrays B, D, and G.

When a gather or scatter instruction is received by the vector control unit, the destination/source register is checked against outstanding arithmetic instructions. If there are no conflicts, the instruction is dispatched to the load/store unit. The load/store unit will then fetch the offset vector register. When this is complete, the vector control unit reissues the instruction and the gather/scatter operation takes place using the previously stored offset vector register to generate the virtual addresses.

A gather instruction is used to collect memory data into vector registers when the memory data does not have a constant stride. The memory data starts with a base address plus an offset number of up to a 64-element (depending on VL) register of offsets. The elements are loaded nearly as fast as a load instruction and are loaded sequentially in the destination register. (The scatter instruction stores the result back to memory using the same offsets.)

3.4.5 Masked Load/Store, Gather/Scatter Instructions

The operation for masked memory instructions is identical to the unmasked versions except the following operations are performed first. The vector controller checks if any outstanding arithmetic instructions will modify the mask register. If not, the vector controller reads the mask from the arithmetic unit and sends it to the load/store unit. The sequence is then performed as above.

3.5 OVERLAP OF ARITHMETIC AND LOAD/STORE INSTRUCTIONS

Arithmetic instructions and load/store instructions may overlap because the functional units are independent. To achieve this overlap, the following conditions must be met:

- The arithmetic instruction must be issued before the load/store instruction.
- There must be no register conflict between the arithmetic and load /store instructions.

In the following example, while the results of vector register 2, V2, are being calculated, vector register 4, V4, is being stored in memory. Consequently, this is referred to as overlapping instructions.

```
VVADDL  V1,V3,V2
VSTL    V4,base,#4
```

In the following examples, an I represents instruction issue time and an E represents instruction execution time. A series of periods represents wait time in the arithmetic unit for deferred instructions. Notice that these are not exact timing examples, since they do not correspond to individual instruction timings, but are for illustration purposes only.

In Example 3–1 the execution of the VLDL instruction does overlap the VVADDL instruction because there is no conflict in the destination vector registers, V3 and V1, for the add and load respectively.

Example 3–1 Overlapped Load and Arithmetic Instructions

```
VVADDL    V1,V2,V3       IEEEEEEEE
VLDL      base,#4,V1      IEEEEEEEEEEEEEE
```

3.5.1 Maximizing Instruction Execution Overlap

Three important hardware features help to maximize instruction overlap in the load/store unit. First, a load or store instruction can execute in parallel with up to two arithmetic instructions, provided the arithmetic instructions are issued first. Second, the chain into store sequence can reduce the perceived execution time of a store instruction. Finally, early detection of no memory faults allows scalar-to-vector communications to overlap with load or store instruction execution.

In the first instruction sequence in Example 3–2 there is little overlapping of instructions, whereas in the second sequence the VVMULL and the second VLDL instructions overlap and require less total time to complete execution. The only difference between the two instruction sequences is the order in which they are issued. Because the VVMULL does not require the result of the second VLDL and can precede that instruction, a significant reduction in execution time is achieved.

Another effective way to maximize the overlap of load/store instructions is to precede, wherever possible, all load and store instructions by at least two arithmetic instructions. In this way both the load/store pipeline and the arithmetic pipeline will be in use.

Example 3–2 Maximizing Instruction Execution Overlap

```
                Instruction Sequence 1

VLDL   base1,#4,V1       IEEEEEEEEE
VLDL   base2,#4,V2                 IEEEEEEEEE
VVMULL V3,V1,V1                            IEEEEE
VVADDL V1,V2,V2                             I....EEEEE
VSTL   V2,base,#4                                IEEEEEEEEE

                Instruction Sequence 2

VLDL   base1,#4,V1       IEEEEEEEEE
VVMULL V3,V1,V1                    IEEEEE
VLDL   base2,#4,V2                  IEEEEEEEEE
VVADDL V1,V2,V2                              IEEEEE
VSTL   V2,base,#4                             IEEEEEEEEE
```

A load instruction cannot begin execution until the register to which it will write is free. A register conflict may occur if the destination register of a load instruction is the same as one of the registers used by a preceding arithmetic instruction. If instruction execution overlap could occur if the load instruction were using a different register, then the register conflict can be eliminated by simply changing the register used.

Example 3–3 shows the effects of register conflict. In the first instruction sequence the VLDL instruction must wait until the VVADDL instruction completes and the VVMULL instruction begins because VLDL will change the contents of one of the registers that provides input to the deferred VVMULL instruction. In the second instruction sequence it is possible to take advantage of the deferred arithmetic instruction queue and overlap the VLDL and arithmetic instruction execution because the VLDL instruction does not change the registers used by the arithmetic instructions. By simply changing the register to which the VLDL will write, the total execution time for the instruction sequence is reduced.

The locality of reference of data plays an important role in determining the performance of load/store operations. Unity stride load and store instructions are the most efficient. For this reason, whenever possible data should be stored in the sequential order in which it is usually referenced.

Example 3–3 Effects of Register Conflict

```
                Instruction Sequence 1

VVADDL    V1,V2,V3      IEEEEE
VVMULL    V1,V2,V4       I....EEEEE
VLDL      base,#4,V1          IEEEEEEEEE

                Instruction Sequence 2

VVADDL    V1,V2,V3       IEEEEE
VVMULL    V1,V2,V4        I....EEEEE
VLDL      base,#4,V5       IEEEEEEEEE
```

Nonunity stride loads and stores can have a significantly higher impact on the performance level of the XMI memory bus as compared to unity stride operations. A far greater number of memory references are required for nonunity stride than is the case for unity stride. If the ratio of cache miss load/store to arithmetic instructions is sufficiently high and nonunity stride is used, bus bandwidth can become the limiting performance factor.

3.6 OUT-OF-ORDER INSTRUCTION EXECUTION

The deferred instruction queue (of length 1) associated with the arithmetic unit allows the vector issue unit to queue one instruction to the arithmetic unit while that unit is still executing a previous instruction. The issue unit checks the status of this queue when it does the functional unit availability check for an instruction. (Both the deferred and currently executing instructions are checked for register availability.) This frees the issue unit to process another instruction rather than having to wait for the arithmetic unit to complete its current instruction.

Example 3–4 shows the use of the deferred arithmetic instruction queue. If a deferred instruction queue was not implemented, the VVMULL instruction could not be issued until the VVADDL was completed (or nearly completed). The VLDL instruction would then not issue until after the VVMULL was issued and would complete much later than in the deferred instruction case. Once the VLDL instruction is issued, no other instructions may be issued. The overlap of instruction execution made possible by the deferred instruction queue can significantly reduce the total execution time. The VLDL instruction can overlap the deferred VVMULL instruction because there are no register conflicts between the two instructions.

Example 3–4 Deferred Arithmetic Instruction Queue

```
                Instruction Sequence

                VVADDL   V1,V2,V3
                VVMULL   V3,V1,V4
                VLDL     base,#4,V2

Execution without Deferred Instruction Queue

Issue VVADDL          IEEEEEEEE
Issue VVMULL                  IEEEEEEEE
Issue VLDL                     IEEEEEEEEEEEEEE

Execution with Deferred Instruction Queue

Issue VVADDL            IEEEEEEEE
Issue deferred VVMULL    I.......EEEEEEEE
Issue VLDL                IEEEEEEEEEEEEEE
```

In Example 3–5 the VLDL instruction cannot begin before VVMULL because VVMULL needs data in V3 before the VLDL takes place.

Example 3–5 A Load Stalled due to an Arithmetic Instruction

```
        VVADDL   V1,V2,V3       IEEEEEEEE
        VVMULL   V3,V4,V5        I.......EEEEEEEE
        VLDL     base,#4,V3             IEEEEEEEEEEEEEE
```

To take advantage of the deferred instruction queue, close attention to instruction ordering and register use is required. Generally, a divide or two other arithmetic instructions should precede each load or store instruction. (In the case of divide instructions, multiple load/store instructions can be overlapped with a single divide instruction.) This is not always possible, since initial loads are usually necessary and there may not be two arithmetic instructions per load/store. Also, some instruction ordering is dictated by the use of the data. But even with these restrictions, it is still important to watch for potential instruction execution overlap.

Example 3–6 is another example of the use of a deferred arithmetic instruction. In this case, a divide instruction is followed by an add and then a load. The deferred instruction queue and the length of the divide instruction combine to "hide" the load instruction (that is, the execution time of the load instruction does not contribute to the total execution time of the instruction sequence). Note also that the divide instruction completes after the load completes. Out of order completion of instructions is possible.

Example 3–6 Use of the Deferred Arithmetic Instruction Queue

```
                Instruction Sequence

                VVDIVL   V1,V2,V3
                VVADDL   V3,V1,V4
                VLDL     base,#4,V5

    Execution without Deferred Instruction Queue

    Issue VVDIVL            IEEEEEEEEEEEEEEEEEEE
    Issue VVADDL                                IEEEEEEEE
    Issue VLDL                                   IEEEEEEEEEEEEEE

    Execution with Deferred Instruction Queue

    Issue VVDIVL            IEEEEEEEEEEEEEEEEEEE
    Issue deferred VVADDL    I..................EEEEEEEE
    Issue VLDL                IEEEEEEEEEEEEEE
```

3.7 CHAINING

Vector operands are generally read from and written to the vector register file. An exception to this process occurs when a store instruction is waiting for the results of a currently executing arithmetic instruction. (Divide instructions are not included in this exception because they do not have the same degree of pipelining as the other instructions.) As results are generated by the arithmetic instruction and are ready to be written to the register file, they are also immediately available for input to the waiting store instruction. Therefore, the store instruction can begin processing the data before the arithmetic instruction has completed. This process is called "chain into store." The store instruction will not overrun the arithmetic instruction because the store instruction cannot process data faster than the arithmetic unit can generate results.

In Example 3–7, the VSTL instruction requires the result of the VVADDL instruction and without chain into store would have to wait for the VVADDL to complete before beginning the store operation. The use of chain into store allows the VSTL operation to begin after the first result of the add is complete, while the VVADDL is still executing and greater overlap of instruction execution is the result. The instruction sequence requires a shorter period of time to complete.

The coordination of the arithmetic operation and the VSTORE for a chain into store is handled by the vector arithmetic unit and depends on a number of factors such as vector length.

Example 3–7 Example of Chain Into Store

```
                Instruction Sequence

                VVADDL   V1,V2,V3
                VVMULL   V1,V2,V4
                VSTL     V3,base,#4

Execution without Chain into Store:

Issue VVADDL            IEEEEEEEE
Issue deferred VVMULL    I.......EEEEEEEE
Issue VSTL                      IEEEEEEEEEEEEEE

Execution with Chain into Store:

Issue VVADDL            IEEEEEEEE
Issue deferred VVMULL    I.......EEEEEEEE
Issue VSTL                IEEEEEEEEEEEEEE
```

3.8 CACHE

With the 1-Mbyte vector cache, up to four load operations with cache misses can be queued at one time. The pipeline continues processing vector element loads until a fourth cache miss occurs. At that point the cache miss queue is full and the pipeline stalls. The pipeline remains stalled until one of the cache misses is serviced. Cache misses on a load instruction degrade the performance of the load/store pipeline.

A cache miss is serviced by a hexword fill. On the XMI, a hexword transfer is 80 percent efficient since one address is sent to receive four quadwords of data. An octaword transfer is 67 percent efficient since one address is sent to receive two quadwords of data. A quadword transfer is only 50 percent efficient since one address is sent to receive one quadword of data. For this reason, stores are more efficient with unity stride than with nonunity stride. A larger piece of memory can be referenced by a single address so that fewer memory references are required.

In the case of load instructions, the comparison of unity and nonunity stride is less straightforward. A nonunity stride cache miss load causes a full hexword to be read from memory even though the load requires only a longword or quadword of data. If the additional data is not referenced by subsequent load instructions, then the nonunity stride load is much less efficient than a unity stride load. If subsequent loads do reference the extra data, then nonunity stride load performance improves due to high cache hit rates for the subsequent loads. For double-precision data there is little degradation due to nonunity stride in this case. For single-precision data, unity stride loads will show significantly higher performance because of the load/store pipeline optimization for single-precision unity stride loads.

3.9 STRIDE/TRANSLATION BUFFER MISS

A vector's stride is the number of memory locations (bytes) between the starting address of consecutive vector elements. A vector with a stride of 1 is contiguous; it has no gaps in memory between vector elements.

Consider the vector arrays A and B in the following DO loop. Vector A has a stride of 1; vector B has a stride of 2.

```
DO 100 I=1,5
   A(I) = B(I*2)
100 CONTINUE
```

When a translation buffer (TB) miss occurs, two PTEs (1 quadword) are fetched from cache. If this fetch results in a cache miss, then a hexword (eight PTEs) is loaded into cache from memory but only two PTEs are installed in the TB.

This handling of TB misses has a large effect on the performance of nonunity stride vectors. A stride of two pages (256 longwords or 128 quadwords) or more can result in a TB miss for each data item. A stride of eight pages (1024 longwords or 512 quadwords) or more can result in a TB miss that can cause a cache miss for each data item. Unity stride is

most efficient in that it runs sequentially through the data and makes full use of all PTEs fetched.

An example of how to avoid large vector strides can be seen in a simple matrix multiplication problem:

```
DO I = 1, N
 DO J = 1, N
  DO K = 1, N
   C(I,J) = C(I,J) + A(I,K)*B(K,J)
  ENDDO
 ENDDO
ENDDO
```

If coded as written, there is a choice of which variable to vectorize on. If the "K" variable is chosen, array A will access FORTRAN rows that are nonunity stride. This choice also means that for every K, a reduction operation is required to sum the product of A and B into the C array. Although reduction functions vectorize, they are less efficient than other methods.

A better choice is to vectorize on either I or J. J is not the best candidate because it involves nonunity stride for both the B and the C arrays. For large values of N, this is an inefficient use of the bus bandwidth, the translation buffer, and the cache. Clearly the optimal solution is to vectorize on the I variable.

Example 3–8 shows a first attempt to code the matrix multiplication in MACRO pseudocode for vectors. Although this example uses unity stride, it is far from optimal. Notice that it is not necessary to load and store C for different values of K because C is dependent only on the I and J variables. By removing the load and store of C from the inner loop, the bytes/FLOP ratio (load and stores: arithmetics) drops from 12 to 2 down to 4 to 2. Example 3–9 shows an improved version.

Example 3–8 Matrix Multiply—Basic

```
        msync  R0               ;synchronize with scalar
LOOP:
        vldl   A(I,K),#4,V0     ;col of A is loaded into V0
        vsmulf B(K,J),V0,V0     ;V0 gets the product of V0
                                ;and the scalar value B(K,J)
        vldl   C(I,J),#4,V1     ;col of C gets loaded into V1
        vvaddf V0,V1,V1         ;V1 gets V0 summed with V1
        vstl   V1,C(I,J),#4     ;V1 is stored back into C

        INC     K               ;increment K by one
        IF (K < N) GOTO LOOP    ;Loop for all values of K

        INC     J               ;increment J by vector length
        IF (J < N) GOTO LOOP    ;Loop for all values of J

        INC     I, RESET J      ;increment I by vector length
        IF (I < N) GOTO LOOP    ;Loop for all values of I

        msync   R0              ;synchronize with scalar
```

Example 3–9 Matrix Multiply—Improved

```
        msync  R0               ;synchronize with scalar
IJLOOP:
        vldl   C(I,J),#4,V1     ;col of C gets loaded into V1
KLOOP:
        vldl   A(I,K),#4,V0     ;col of A is loaded into V0
        vsmulf B(K,J),V0,V0     ;V0 gets the product of V0
                                ;and the scalar value B(K,J)
        vvaddf V0,V1,V1         ;V1 gets V0 summed with V1

        INC     K               ;increment K by one
        IF (K < N) GOTO KLOOP   ;Loop for all values of K

        vstl   V2,C(I,J),#4     ;V2 is stored back into C

        INC     J, RESET K      ;increment J by vector length
        IF (J < N) GOTO IJLOOP  ;Loop for all values of J

        INC     I, RESET J      ;increment I by vector length
        IF (I < N) GOTO IJLOOP  ;Loop for all values of I

        msync   R0              ;synchronize with scalar
```

3.10 REGISTER REUSE

The concept used in Example 3–9 to reuse the data when it has already been loaded into a vector register is known as register reuse. Register reuse can be extended further by using all available vector registers to decrease the bytes/FLOP ratio and improve performance. With maximum register reuse, programs on the VAX 6000 Model 400 vector processor can approach a peak single-precision performance of 90 MFLOPs and a peak double-precision performance of 45 MFLOPs.

To implement register reuse for matrix multiply, the J loop must be unrolled. By precomputing 14 partial results, using only the first column of A with 14 different columns of B, it is possible to use 14 vector registers (instead of 14 memory locations) to hold the partial results. Thus, all N rows of B can be accessed in groups of 14 columns to compute the first 14 columns of C. When the final row of B is reached, the results are chained into a store into array C. Then the next set of 14 columns of C will be calculated. The unrolling depth of 14 is chosen because of the number of vector registers. Example 3–10 shows the MACRO pseudocode to accomplish this for values of N <= 64. Although the code length is longer, the performance is greatly improved by the segments of code that are purely vector arithmetics. The bytes/FLOP ratio has dropped to better than 4 to 14, allowing the algorithm to approach peak vector speeds. When implemented in matrix solvers, speedups greater than 25 have been realized in a VAX 6000 Model 410 vector processor computer system.

Example 3–10 Matrix Multiply—Optimal

```
        msync  R0                   ;synchronize with scalar
        mtvlr #N                    ;
                                    ;
loop2:                              ;first segment
                                    ;
        vldl       A(I,K),#4,v0     ;
                                    ;mul
                                    ;
        vsmulf     B(K,J),v0,v2     ;

        vsmulf     B(K,J+1),v0,v3   ;
        vsmulf     B(K,J+2),v0,v4   ;
        vsmulf     B(K,J+3),v0,v5   ;
        vsmulf     B(K,J+4),v0,v6   ;
        vsmulf     B(K,J+5),v0,v7   ;
        vsmulf     B(K,J+6),v0,v8   ;
        vsmulf     B(K,J+7),v0,v9   ;
        vsmulf     B(K,J+8),v0,v10  ;
        vsmulf     B(K,J+9),v0,v11  ;
        vsmulf     B(K,J+10),v0,v12 ;
        vsmulf     B(K,J+11),v0,v13 ;
        vsmulf     B(K,J+12),v0,v14 ;
        vsmulf     B(K,J+13),v0,v15 ;
                                    ; update
                                    ;
        INC  K
 loop1:                             ;
                                    ;load col of A
                                    ;
        vldl       A(I,K),#4,v0     ;
                                    ;mul and add
                                    ;
        vsmulf     B(K,J),v0,v1
        vvaddf     v1,v2,v2         ;
        vsmulf     B(K,J+1),v0,v1
        vvaddf     v1,v3,v3         ;
        vsmulf     B(K,J+2),v0,v1
        vvaddf     v1,v4,v4         ;
        vsmulf     B(K,J+3),v0,v1
```

Example 3–10 Cont'd on next page

Example 3–10 (Cont.) Matrix Multiply—Optimal

```
                vvaddf      v1,v5,v5          ;
                vsmulf      B(K,J+4),v0,v1
                vvaddf      v1,v6,v6          ;
                vsmulf      B(K,J+5),v0,v1
                vvaddf      v1,v7,v7          ;
                vsmulf      B(K,J+6),v0,v1
                vvaddf      v1,v8,v8          ;
                vsmulf      B(K,J+7),v0,v1
                vvaddf      v1,v9,v9          ;
                vsmulf      B(K,J+8),v0,v1
                vvaddf      v1,v10,v10        ;
                vsmulf      B(K,J+9),v0,v1
                vvaddf      v1,v11,v11        ;
                vsmulf      B(K,J+10),v0,v1
                vvaddf      v1,v12,v12        ;
                vsmulf      B(K,J+11),v0,v1
                vvaddf      v1,v13,v13        ;
                vsmulf      B(K,J+12),v0,v1
                vvaddf      v1,v14,v14        ;
                vsmulf      B(K,J+13),v0,v1
                vvaddf      v1,v15,v15        ;
                                              ; update
                                              ;
                INC  K
                IF (K < N) GOTO LOOP1         ;Loop for all values of K
                                              ;
                                              ;last element
loopa1:                                       ;
                                              ;load col of A
                                              ;
                vldl        A(I,K),#4,v0      ;
                                              ;mul, add and store
                                              ;
```

Example 3–10 Cont'd on next page

Example 3–10 (Cont.) Matrix Multiply—Optimal

```
                vsmulf      B(K,J),v0,v1
                vvaddf      v1,v2,v2
                vstl        v2,C(I,J),#4        ;
                vsmulf      B(K,J+1),v0,v1
                vvaddf      v1,v3,v3
                vstl        v3,C(I,J+1),#4      ;
                vsmulf      B(K,J+2),v0,v1
                vvaddf      v1,v4,v4
                vstl        v4,C(I,J+2),#4      ;
                vsmulf      B(K,J+3),v0,v1
                vvaddf      v1,v5,v5
                vstl        v5,C(I,J+3),#4      ;
                vsmulf      B(K,J+4),v0,v1
                vvaddf      v1,v6,v6
                vstl        v6,C(I,J+4),#4      ;
                vsmulf      B(K,J+5),v0,v1
                vvaddf      v1,v7,v7
                vstl        v7,C(I,J+5),#4      ;
                vsmulf      B(K,J+6),v0,v1
                vvaddf      v1,v8,v8
                vstl        v8,C(I,J+6),#4      ;
                vsmulf      B(K,J+7),v0,v1
                vvaddf      v1,v9,v9
                vstl        v9,C(I,J+7),#4      ;
                vsmulf      B(K,J+8),v0,v1
                vvaddf      v1,v10,v10
                vstl        v10,C(I,J+8),#4     ;
                vsmulf      B(K,J+9),v0,v1
                vvaddf      v1,v11,v11
                vstl        v11,C(I,J+9),#4     ;
                vsmulf      B(K,J+10),v0,v1
                vvaddf      v1,v12,v12
                vstl        v12,C(I,J+10),#4    ;
                vsmulf      B(K,J+11),v0,v1
                vvaddf      v1,v13,v13
                vstl        v13,C(I,J+11),#4    ;
                vsmulf      B(K,J+12),v0,v1
                vvaddf      v1,v14,v14
                vstl        v14,C(I,J+12),#4    ;
                vsmulf      B(K,J+13),v0,v1
                vvaddf      v1,v15,v15
                vstl        v15,C(I,J+13),#4    ;
                                                ; update
                                                ;
```

Example 3–10 Cont'd on next page

Example 3–10 (Cont.) Matrix Multiply—Optimal

```
                RESET K
                RESET I
                INC  J by 14
                IF (J < N) GOTO LOOP2        ;Loop for all values of K

                msync  R0                    ;synchronize with scalar
```

A Algorithm Optimization Examples

This appendix illustrates how the characteristics of the VAX 6000 series vector processor can be used to build optimized routines for this system and how the algorithm and its implementation can change the performance of an application on the VAX 6000 processor.

The VAX 6000 series vector processor delivers high performance for computationally intensive applications. Based on CMOS technology, the VAX 6000 Model 400 vector processor is capable of operating at peak speeds of 90 MFLOPs single precision and 45 MFLOPs double precision. Linear algebra and signal processing applications that utilize the various hardware features have demonstrated vector speedups between 3 and 35 over the scalar VAX 6000 CPU times. With the integrated vector processing available on the VAX 6000 series, the performance of computationally intensive applications may now approach that of supercomputers.

Algorithm changes can alter the data access patterns to more efficiently use the memory subsystem, can increase the average vector length, and can minimize the number of vector operations required. By applying Amdahl's Law of vectorization, performance can be improved by increasing the percentage of code that is vectorized.

Four basic optimization methods that take advantage of the processing power of VAX 6000 series system include:

- Rearrange code for maximum vectorization of the inner loop and remove data dependencies within the loop
- Vectorize across contiguous memory locations to produce unity stride vectors for increased cache hit rates and optimized cache miss handling
- Reuse the data already loaded into the vector registers as frequently as possible to reduce the number of vector load and store operations
- Maximize instruction execution overlap by pairing arithmetic instructions between load and store instructions wherever possible

Further information on optimization techniques in FORTRAN can be found in the *VAX FORTRAN Performance Guide* available with the FORTRAN-High Performance Option.

Two groups of applications that have high vector processing potential include equation solvers and signal processing routines. For example, computational fluid dynamics, finite element analysis, molecular dynamics, circuit simulation, quantum chromodynamics, and economic modeling applications use various types of simultaneous or differential equation solvers. Applications such as air pollution modeling, seismic analysis, weather forecasting, radar imaging, speech and image processing, and many other scientific and engineering applications use signal processing routines, such as fast Fourier transforms (FFT), to obtain solutions.

A.1 EQUATION SOLVERS

Equation solvers generally fall into four categories: general rectangle, symmetric, hermitian, and tridiagonal. The most common benchmark used to measure a computer system's ability to solve a general rectangular system of linear equations is Linpack. The Linpack benchmarks, developed at Argonne National Laboratory, measure the performance across different computer systems while solving dense systems of 100, 300, and 1000 linear equations.

These benchmarks are currently written to call subroutines from the Linpack library. The subroutines, in turn, call the basic linear algebra subroutines (BLAS) at the lowest level. For each benchmark size, there are different optimization rules which govern the type of changes permitted in the Linpack report. Optimizations to the BLAS routines are always allowed. Modifications can be made to the FORTRAN source or by supplying the routine in macrocode. Algorithm changes are only allowed for the largest problem size, the solution to a system of 1000 linear equations.

The smallest problem size uses a two-dimensional array that is 100 by 100. The benchmarks are written to use Gaussian elimination for solving 100 simultaneous equations. This two-step method features a factorization routine, xGEFA, and a solver, xGESL. Both are column-oriented algorithms and use vector-vector level 1 BLAS routines. Column orientation increases program efficiency because it improves locality of data based on the way FORTRAN stores arrays.

As shown in Example A–1, the BLAS level 1 routines allow the user to schedule the instructions optimally in vector macrocode. Deficiencies in BLAS 1 routines include frequent synchronization, a large calling overhead, and more vector load and store operations in comparison to other vector arithmetic operations.

Example A–1 Core Loop of a BLAS 1 Routine Using Vector-Vector Operations

```
xAXPY - computes Y(I) = Y(I) + aX(I)
        where x = precision = F, D, G

        MSYNC                       ;synchronize with scalar
LOOP:
        VLDx    X(I),std,VR0        ;X(I) is loaded into VR0
        VSMULx  a,VR0,VR0           ;VR0 gets the product of VR0
                                    ;and the scalar value "a"
        VLDx    Y(I),std,VR1        ;Y(I) get loaded into VR1
        VVADDx  VR0,VR1,VR1         ;VR1 gets VR0 summed with VR1
        VSTx    VR1,Y(I),std        ;VR1 is stored back into Y(I)

        INC     I                   ;increment I by vector length
        IF (I < SIZ) GOTO LOOP      ;Loop for all values of I
        MSYNC                       ;synchronize with scalar
```

The performance of the Linpack 100 by 100 benchmark, which calls the routine in Example 3–7 showing execution without chain into store, shows how an algorithm with approximately 80 percent vectorization can be limited by the scalar portion. One form of Amdahl's Law relates the percentage of vectorized code compared to the percentage of scalar code to define an overall vector speedup. This ratio between scalar runtime and vector runtime is described by the following formula:

$$\text{Vector Speedup} = \frac{\text{Time Scalar}}{(\%\text{scalar}*\text{Time Scalar})) + (\%\text{vector}*\text{Time Vector})}$$

Under Amdahl's Law, the maximum vector speedup possible, assuming an infinitely fast vector processor, is:

$$\text{Vector Speedup} = \frac{1.0}{(.2)*1.0 + (.8)*0} = \frac{1.0}{0.2} = 5.0$$

As shown in Figure A–1, the Model 400 processor achieves a vector speedup of approximately 3 for the 100 by 100 Linpack benchmark when using the BLAS 1 subroutines. It follows Amdahl's Law closely because it is small enough to fit the vector processor's 1-Mbyte cache and, therefore, incurs very little overhead due to memory hierarchy.

Figure A–1 Linpack Performance Graph, Double-Precision BLAS Algorithms

Refer to the printed version of this book, EK–60VAA–PG.

For the Linpack 300 by 300 benchmark, optimizations include the use of routines that are equivalent to matrix-vector level 2 BLAS routines. Example A–2 details the core loop of a BLAS 2 routine. BLAS 2 routines make better use of cache and translation buffers than the BLAS 1 routines do. Also, BLAS 2 routines have a better ratio between vector arithmetics and vector load and stores. The larger matrix size increases the average vector length. Performance is improved by amortizing the time to decode instructions across a larger work load.

By removing one vector load and one vector store from the innermost loop, the BLAS 2 routine has a better ratio of arithmetic operations to load and store operations than BLAS 1 routines. Although the 300 by 300 array fits into the vector processor's 1-Mbyte cache, not all the cache can be mapped by its translation buffer. By changing the sequence in which this routine is called in the program, the data access patterns can be altered to better use the vector unit's translation buffer. Thus, higher performance is obtained.

The percent of vectorization increases primarily because of the increase in the matrix size from 100 by 100 to 300 by 300. With a vector fraction of approximately 95 percent, Figure A–1 shows the speedup improvement in the 300 by 300 benchmark when using methods based on BLAS 2 routines. With a matrix vector algorithm, the 300 by 300 benchmark yields speedups of between 10 and 12 over its scalar counterpart.

Example A–2 Core Loop of a BLAS 2 Routine Using Matrix-Vector Operations

```
xGEMV - computes Y(I) = Y(I) + X(J)*M(I,J)
        where x = precision = F, D, G

        MSYNC                       ;synchronize with scalar
ILOOP:
        VLDx    Y(I),std,VR0        ;Y(I) is loaded as VR0
JLOOP:
        VLDx    M(I,J),std,VR1      ;VR1 gets columns of M(I,J)
        VSMULx  X(J),VR1,VR2        ;VR2 gets the product of VR1
                                    ;and X(J) as a scalar
        VVADDx  VR0,VR2,VR0         ;VR0 gets VR0 summed with VR2
        INC     J
        IF (J < SIZ) GOTO JLOOP     ;Loop for all values of J

        VSTx    VR0,Y(I),std        ;VR0 gets stored into Y(I)
        INC     I
        IF (I < SIZ) GOTO ILOOP     ;Loop for all values of I
        MSYNC                       ;synchronize with scalar
```

There are no set rules to follow when solving the largest problem size, a set of 1000 simultaneous equations. One potential tool for optimizing this benchmark is the LAPACK library, developed by Argonne National Laboratory in conjunction with the University of Illinois Center for Supercomputing Research and Development (CSRD). The LAPACK library features equation-solving algorithms that will block the data array into sections that fit into a given cache size. The LAPACK library calls not only the BLAS 1 and BLAS 2 routines but also a third level of BLAS, called matrix-matrix BLAS or the BLAS level 3.

Example A–3 shows that a matrix-matrix multiply is at the heart of one BLAS 3 routine. The matrix multiplication computation can be blocked for modern architectures with cache memories. Highly efficient vectorized matrix multiplication routines have been written for the VAX vector architecture. For example, a double precision 64 by 64 matrix multiplication achieves over 85 percent of the peak MFLOPS on the Model 400 system.

Performance can be further improved with other methods that increase the reuse of data while it is contained in the vector registers. For example, loop unrolling can be done until all the vector registers have been fully utilized. Partial results can be formed within the innermost

Example A–3 Core Loop of a BLAS 3 Routine Using Matrix-Matrix Operations

```
xGEMM - computes Y(I,J) = Y(I,J) + X(I,K)*M(K,J)
        here x = precision = F, D, G

        MSYNC                   ;synchronize with scalar
IJLOOP:
        VLDx    Y(I,J),std,VR0  ;Y(1:N,J) gets loaded into VR0
KLOOP:
        VLDx    M(K,J),std,VR1  ;K(1:N,K) get loaded into VR1
        VSMULx  X(I,K),VR1,VR1  ;VR1 gets VR1 summed with
                                ;X(I,K) as a scalar
        VVADDx  VR0,VR2,VR0     ;VR0 gets VR0 summed with VR2
        INC     K               ;increment K by vector length
        IF (K < SIZ) GOTO KLOOP

        RESET   K               ;reset I to SIZ
        VSTx    VR0,Y(I,J),std  ;VR0 gets stored into Y(I,J)
        INC     I               ;increment I by vector length
        IF (I < SIZ) GOTO IJLOOP
        INC     J               ;increment J by vector length
        RESET   I               ;reset I to SIZ
        IF (J < SIZ) GOTO IJLOOP
        MSYNC                   ;synchronize with scalar
```

loop to minimize the loads and stores required. Because both rows and columns are traversed, the algorithm can be blocked for cache size. The VAX 6000 Model 400 exhibits vector speedups greater than 35 for the 64 by 64 matrix multiplication described above.

Although the overall performance of the 1000 by 1000 size benchmark is less than a single 64 by 64 matrix multiplication, it does indicate the potential performance when blocking is used. Improving the performance of this benchmark is most challenging because the 1000 by 1000 matrix requires about eight times the vector cache size of 1 Mbyte. Further analysis is being conducted to determine the most efficient block size to use, that would maximize the use of BLAS 3 and remain within the size of the cache for a given block of code.

The vectorized fraction increases to approximately 98 percent for the 1000 by 1000 benchmark. The proportion of vector arithmetics relative to vector load and stores is much improved for the BLAS 3s. Although the cache is exceeded, performance more than doubles when using a method that can block data based on the BLAS 3 algorithms. Therefore, the performance of the VAX 6000 Model 400 on the blocked Linpack 1000

by 1000 obtained a vector speedup of approximately 25, as shown in Figure A–1.

A.2 SIGNAL PROCESSING—FAST FOURIER TRANSFORMS

The Fourier transform decomposes a waveform, or more generally, a collection of data, into component sine and cosine representation. The discrete Fourier transform (DFT) of a data set of length *N* performs the transformation following the strict mathematical definition which requires *O(N**2)* floating-point operations. The fast Fourier transform (FFT), developed by Cooley and Tukey in 1965, reduced the number of operations to *O(N x LOG[N])*, improving computational speed significantly.

Figure A–2 shows that the complex data in the bottom butterfly is multiplied in each stage by the appropriate weight. The result is then added to the top butterfly and subtracted from the bottom butterfly. If the algorithm is left in this configuration, it must use nonunity stride vectors, very short vectors, or masked arithmetic operations to perform the very small butterflies.

A.2.1 Optimized One-Dimensional Fast Fourier Transforms

The bit-reversal process that permutes the data to a form that enables the Cooley-Tukey algorithm to work is also shown in Figure A–2. When using vectors, a common approach to performing the bit-reversal reordering is to use vector gather or scatter instructions. These instructions allow vector loads and stores to be performed using an index register. Vector loads and stores require a constant stride. However, vector gather and scatter operations allow the user to build a vector of offsets to support indirect addressing in vector mode. Both gather and scatter instructions are available with VAX vectors.

A vector implementation of the FFT algorithm has been developed that is well suited for the VAX vector architecture. One optimization made to the algorithm involves moving the bit-reversal section of the code to a place where the data permutation will benefit vector processing. By doing so, two goals are accomplished. First, the slower vector gather operations are moved to the center of the algorithm such that the data will already be in the vector cache. In Figure A–2, the first FFT stage starts out with large butterfly distances. After each stage the butterfly distance is halved. For the optimized version shown in Figure A–3, the bit-reversal permutation

Figure A–2 Cooley-Tukey Butterfly Graph, One-Dimensional Fast Fourier Transform for N = 16

Refer to the printed version of this book, EK–60VAA–PG.

is performed as close to the center as possible, when the *stage number = LOG(N)/2*. To complete the algorithm, the butterfly distances now increase again. Second, this process entirely eliminates the need for short butterflies.

Another optimization made to the FFT algorithm is the use of a table lookup method to access the sine and cosine factors, which reduces repetitive calls to the computationally intensive trigonometric functions. The initialization of this trigonometric table has been fully vectorized but shows only a modest factor of 2 performance gain. To build the table, a first order linear recurrence loop is formed that severely limits vector speedup. Because this calculation is only done once, it becomes negligible for multiple calls to the one-dimensional FFTs and for all higher dimensional FFTs. The benchmark shown in Figure A–4 was looped and includes the calculation of the trigonometric table performed once for each FFT data length.

Figure A–3 Optimized Cooley-Tukey Butterfly Graph, One-Dimensional Fast Fourier Transform for N = 16

Refer to the printed version of this book, EK–60VAA–PG.

Reusing data in the vector registers also saves vector processing time. The VAX vector architecture provides 16 vector registers. If all 16 registers are used carefully, data can be reused by two successive butterfly stages without storing and reloading the data. With half the number of loads and stores, the vector performance almost doubles.

A.2.2 Optimized Two-Dimensional Fast Fourier Transforms

The optimized one-dimensional FFT can be used to compute multidimensional FFTs. Figure A–5 shows how an *N* by *N* two-dimensional FFT can be computed by performing *N* one-dimensional column FFTs and then *N* one-dimensional row FFTs. The same routine can be called for column or row access FFTs by simply varying the stride parameter that is passed to the routine. (Note: In FORTRAN, the column

Figure A–4 One-Dimensional Fast Fourier Transform Performance Graph, Optimized Single-Precision Complex Transforms

Refer to the printed version of this book, EK–60VAA–PG.

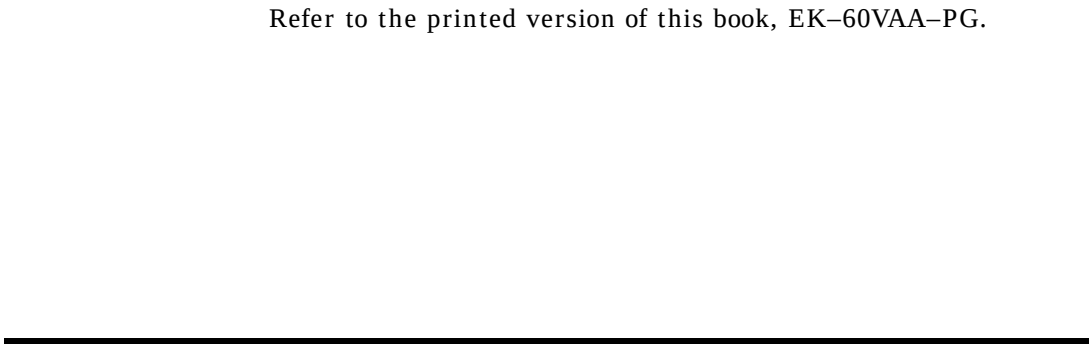
Figure A–5 Two-Dimensional Fast Fourier Transforms Using N Column and N Row One-Dimensional Fast Fourier Transforms

Refer to the printed version of this book, EK–60VAA–PG.

access is unity stride and the row access has a stride of the dimension of the array.)

For improved performance on VAX vector systems, the use of a matrix transpose can dramatically increase the vector processing performance of two-dimensional FFTs for large values of *N* (that is, *N* > 256). The difference between unity stride and nonunity stride is the key performance issue. Figure A–6 shows that a vectorized matrix transpose can be performed after each set of *N* one-dimensional FFTs. The computation will be equivalent to Figure A–2 but with a matrix transpose: each one-dimensional FFT will be column access which is unity stride. The overhead of transposing the matrix becomes negligible for large values of *N*.

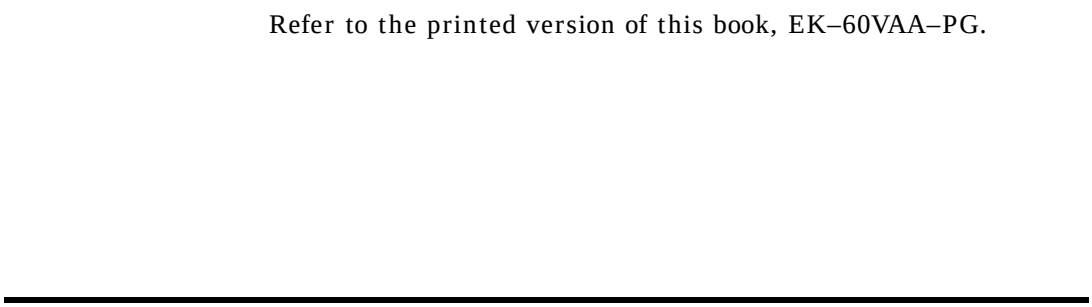
Figure A–6 Two-Dimensional Fast Fourier Transforms Using a Matrix Transpose Between Each Set of N Column One-Dimensional Fast Fourier Transforms



When the value of *N* is relatively small (that is, *N* < 256), the two-dimensional FFT can be computed by calling a one-dimensional FFT of length *N**2*. The small two-dimensional FFT can achieve performance equal to that of the aggregate size one-dimensional FFT by linearizing the data array. Figure A–7 shows the tradeoff between using the linearized two-dimensional routine (for small *N*) and the transposed method (for large *N*) to maintain high performance across all data sizes.

The optimization of an algorithm that vectorizes poorly in its original form has been shown. The resulting algorithm yields much higher performance on the VAX 6000 Model 400 processor. High performance is due to the unique way the algorithm touches contiguous memory locations and its effort to maximize the vector length. The implementation described above always uses unity stride vectors and always results in a vector length of 64 for FFT lengths greater than 2K (2 x 1024).

Figure A–7 Two-Dimensional Fast Fourier Transform Performance Graph, Optimized Single-Precision Complex Transforms



Glossary

accumulator: A register that accumulates data for arithmetic or logic operations.

ALU: Arithmetic Logic Unit, a subset of the operation instruction function unit that performs arithmetic and logical operations, usually in binary form.

Amdahl's Law: A mathematical equation that states that a system is dominated by its slowest process.

arithmetic exception: A software error that occurs while performing an arithmetic or floating point operation.

array: Elements or data arranged in rows and columns.

array processor: A vector processor consisting of auxiliary hardware attached to a host CPU. It is typically attached by an I/O bus and is treated by the host as a foreign I/O device. Also called an *attached vector processor*.

asynchronous: Pertaining to events that are scheduled without any specific time reference; not synchronized to a master clock. For example, while performing arithmetic operations, the vector processor operates by an *asynchronous* schedule to that of the scalar processor, which is free to perform other operations.

benchmark: A program used to evaluate the performance of a computer for a given application.

cache miss: The case when the processor cannot find an item in the cache that it needs to perform an operation; also called a *cache fault*. When this happens, the item is fetched from main memory at the slower main memory speed.

chaining: A form of instruction overlap that uses a special hardware path to load the output of one function unit directly into the input of a second function unit, as well as into the destination register of the first instruction.

concurrent: Occurring during the same interval of time; may or may not be simultaneous.

control word operand: The portion of the instruction that indicates which registers to use and enables or disables certain functions.

crossover point: The vector length at which the vector processor's performance exceeds that of the scalar processor.

data dependency: The case when data for one arithmetic instruction depends on the result of a previous instruction so both instructions cannot execute at the same time.

decomposition: Part of the compilation process that prepares code for parallel or vector processing. It includes dependency analysis, recognition of parallel or vector inhibitors, and code restructuring. Decomposition can be automatic (controlled entirely by the compiler), directed (controlled by compiler directives or statements in the source code), or by a combination of the two.

dependency analysis: An evaluation of how data flows through memory. The analysis is performed to identify data dependencies in a program unit and to determine the requirements they place on decomposition.

first-order linear recurrence: A cyclic data dependency in a linear function that involves one variable.

function unit: A section of the vector processor (or any other processor) that performs a specific function and operates independently from other units. Typically, a function unit performs related operations; for example, an add function unit may also perform subtraction since the operations are similar. Also called *pipe*.

gather: The collection of data from memory starting with a base address plus an offset address; the instruction that performs this action placing the data into sequential locations in a vector register.

integrated vector processor: A vector processor consisting of a coprocessor that is tightly coupled with its host scalar processor.

interleaving: Using multiple memory boards in main memory so that one or more processors can access data, that is distributed among different boards, concurrently.

IOTA: An instruction to generate a compressed vector of offset addresses for use in a gather or scatter instruction.

latency: The time that elapses from an element entering a pipeline until the first result is produced.

linear recurrence: A cyclic data dependency in a linear function.

LINPACK: An industry-wide benchmark used to measure the performance characteristics of various computers. Unlike some benchmarks, LINPACK is a real application package doing linear algebra calculations.

Livermore FORTRAN kernels: A set of loops used to measure the performance characteristics of various computers and the efficiency of vectorizing compilers. The 24 kernels are fragments of programs from scientific and engineering applications. Also known as Livermore Loops.

loop fusion: A transformation that takes two separate DO loops and makes a single DO loop one out of them.

loop rolling: A transformation that combines parts of DO loops to allow for vectorization.

loop unrolling: A transformation that separates certain DO loops into smaller loops to reduce overhead.

loop-carried dependency: A data dependency that crosses iteration boundaries or that crosses between a loop and serial code. The dependency would not exist in the absence of the loop.

loop-independent dependency: A data dependency that occurs whether or not a loop iterates.

mask register: A vector control register used to select the elements of a vector that will participate in a particular operation.

megaflops: A measure of the performance of a computer—the rate at which the computer executes floating-point operations. Expressed in terms of "millions of floating-point operations per second." Known as MFLOPS.

memory bandwidth: The range of speeds at which a memory bus or path can carry data. The lowest speed is usually 0 (no data) and is, therefore normally not mentioned. For example, a memory bus that can provide data at speeds from 0 to 512 megabytes per second is said to have a bandwidth of 512 Mbytes/s.

memory bank: An individual memory board. Groups of memory banks make up interleaved high-speed main memory.

MFLOPS: See *megaflops*.

MIPS: A measure of the performance of a computer—the rate at which the computer executes instructions. Expressed in terms of "millions of instructions per second."

MTF: Match true/false: When masked operations are enabled, only elements for which the Vector Mask Register bit matches true (or false, depending on the condition) are operated upon.

optimizer: A program that scans code and changes the sequence or placement of the code to allow it to run faster and more efficiently on the scalar or vector processor. The program also reports dependencies that it cannot resolve.

overlapping: Executing two or more instructions so part of their execution occurs at the same time. For example, while one instruction is performing an arithmetic operation, another is storing results to memory.

parallelization: The part of the compilation process that prepares a section of source code for parallel processing.

parallel processing: Concurrent execution of multiple segments from the same program image.

peak MFLOPS: The theoretical maximum number of floating-point operations per second; a vector processor's best attainable performance.

pipe, or pipeline: A section of a processor that performs a specific function. For example, one pipe can be used for addition and subtraction, one for multiply, and one for load/store operations. Each operates independently from the others. Also called *function unit*.

pipeline length: The number of segments of a pipeline within one function unit, which is the limit of the number of elements that can be executed in that function unit at one time.

pipelining: A technique used in high-performance processors whereby a stream of data is processed in contiguous subtasks at separate stations along the pipeline.

recursion: A process in which one of its steps makes use of the results of steps from an earlier statement. Also called *recurrence*.

scalar: A single element or number.

scalar operand: The symbolic expression representing the scalar data accessed when an operation is executed; for example, the input data or arguement.

scatter: The process of storing data into memory starting with a base address plus an offset address; the instruction that performs this action of placing the data back into memory.

second-order linear recurrence: Two cyclic data dependencies occurring in a linear function.

speedup ratio: The vector processor performance divided by the scalar processor performance; indicates how many times faster the computer is with a vector processor installed than without it.

store: To move data from vector register to memory; for example, the VSTx command moves the results from the vector register back to memory.

stride: The number of memory locations (bytes) between the starting address of consecutive vector elements; for example: A vector has a starting address, a length of 10 elements, and a stride of 4 bytes between the start of each element.

stripmining: The process of splitting a vector into two or more subvectors, each of which will fit in a vector register, so each is processed sequentially by the vector procesor.

sustained megaflops: The average floating-point performance achieved on a computer during some reference application (or benchmark).

translation buffer: A hardware or software mechanism to remember successive virtual address translations and virtual page addresses, used to save time when referencing memory locations on the same memory page.

translation buffer miss: An occurrence where the processor cannot translate the virtual address using the current contents of the translation buffer. In such a case, the processor is forced to load new information into the translation buffer to furnish the address.

unknown dependency: An unclear relationship between multiple references to a memory location; a relationship in which the final value of the location may or may not depend on serial execution of the code involved.

vector: A data structure composed of scalar elements with the same data type and organized as a simple linear sequence.

vector instruction: A native computer instruction that recognizes a vector as a native data structure and that can operate on all the elements of a vector concurrently.

vector length register (VLR): A 7-bit register that controls the number of elements used in the vector registers, from 0 to 64.

vector load: To move data from memory to the vector registers. For example, the VLDx command moves data to the vector registers.

vector mask register (VMR): A 64-bit register that enables and disables the use of individual elements within a vector register.

vector operand: A string of scalar data items with the same data type that are processed in a single operation.

vector processor: A processor that operates on vectors, making use of pipelines that overlap key functional operations and performing the same operations repeatedly, to achieve high processing speeds.

vector processing: Execution of vector operations on a vector processor. A single vector operation is capable of modifying each element in a vector operand concurrently.

vector register: A high-speed buffer contained in the vector processor's CPU, consisting of word- or address-length bit sequences that are directly accessible by the processor. A VAX vector register can hold 64 elements of 64 bits each.

vectorizable: Capable of being converted to code that can be processed on a vector processor.

vectorization: Part of the compilation process that prepares a section of source code for vector processing.

vectorization factor: In a program, the fraction of code that can be converted to run on a vector processor. For example, a program with a *vectorization factor* above 70% will perform well on a system that has a vector processor.

vector-scalar operation: An operation such as add, subtract, and so forth, in which a scalar number operates with each element of a vector register and places results in matching elements of another vector register.

vector-vector operation: An operation in which each element of one vector register operates with the corresponding element of a second vector register and then places the results in matching elements of a third vector register.

Index

L

M

N

O

P

Q

R