CF77™ Compiling System, Volume 3: Vectorization Guide

SG–3073 5.0

Cray Research, Inc.

Because of space restrictions, the following abbreviations are used in place of specific system names:

CX	Includes all models of CRAY X-MP computer systems
CEA	Includes all models of the Extended Architecture (EA) series, including CRAY Y-MP and CRAY X-MP EA computer systems
CRAY-2	Includes all models of CRAY-2 computer systems
CX/CEA	Includes all models of CRAY X-MP computer systems plus all models of CRAY Y-MP and CRAY X-MP EA computer systems

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120

Order desk (612) 681–5907
Fax number (612) 681–5920

Record of Revision

The date of printing or software version number is indicated in the footer. In reprints with revision, changes are noted by revision bars along the margin of the page.

Version	***Description***
5.0	August 1991 – Original printing.

Preface

This manual is explains the use of vector processing with the CF77 compiling system, a product of Cray Research Inc. (CRI). The compiling system operates on all Cray Research computers and operating systems. This manual is part of a set for the compiling system, which includes the following titles:

- *CF77 Compiling System Ready Reference*, publication SQ–3070
- *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071
- *CF77 Compiling System, Volume 2: Compiler Message Manual*, publication SR–3072
- *CF77 Compiling System, Volume 4: Parallel Processing Guide*, publication SG–3074
- *UNICOS I/O Technical Note*, publication SN–3075

The text assumes that you are a user of CF77 and have access to these other publications but does not assume previous familiarity with vector processing.

Hardware conventions

The Hardware Product Line sheet, located at the end of this preface, defines the hardware naming conventions used in this manual. This sheet shows both the chronological evolution of CRI mainframes and the characteristics of each mainframe group. The reverse side of the sheet contains definitions of the terms used on the sheet and throughout this manual.

The following reference conventions are used throughout this manual:

Convention	Meaning
command(1)	The designation (1) following a command name indicates that the command is documented in *UNICOS User Commands Reference Manual*, publication SR–2011
routine(3*x*)	The designation (3*x*) following a routine name indicates that the routine is documented in one of the CRI library reference manuals (SR–2079, SR–2080, SR–2081, SR–2082, or SR–2057). The letter following the number 3 indicates the appropriate manual.

Reader comments

If you have comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Call our Software Information Services department at (612) 683–5729.
- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to Internet, using one of the following Internet addresses:

  `pubs3073@timbuk.cray.com` (comments specific to this manual)

  `publications@timbuk.cray.com` (general comments)

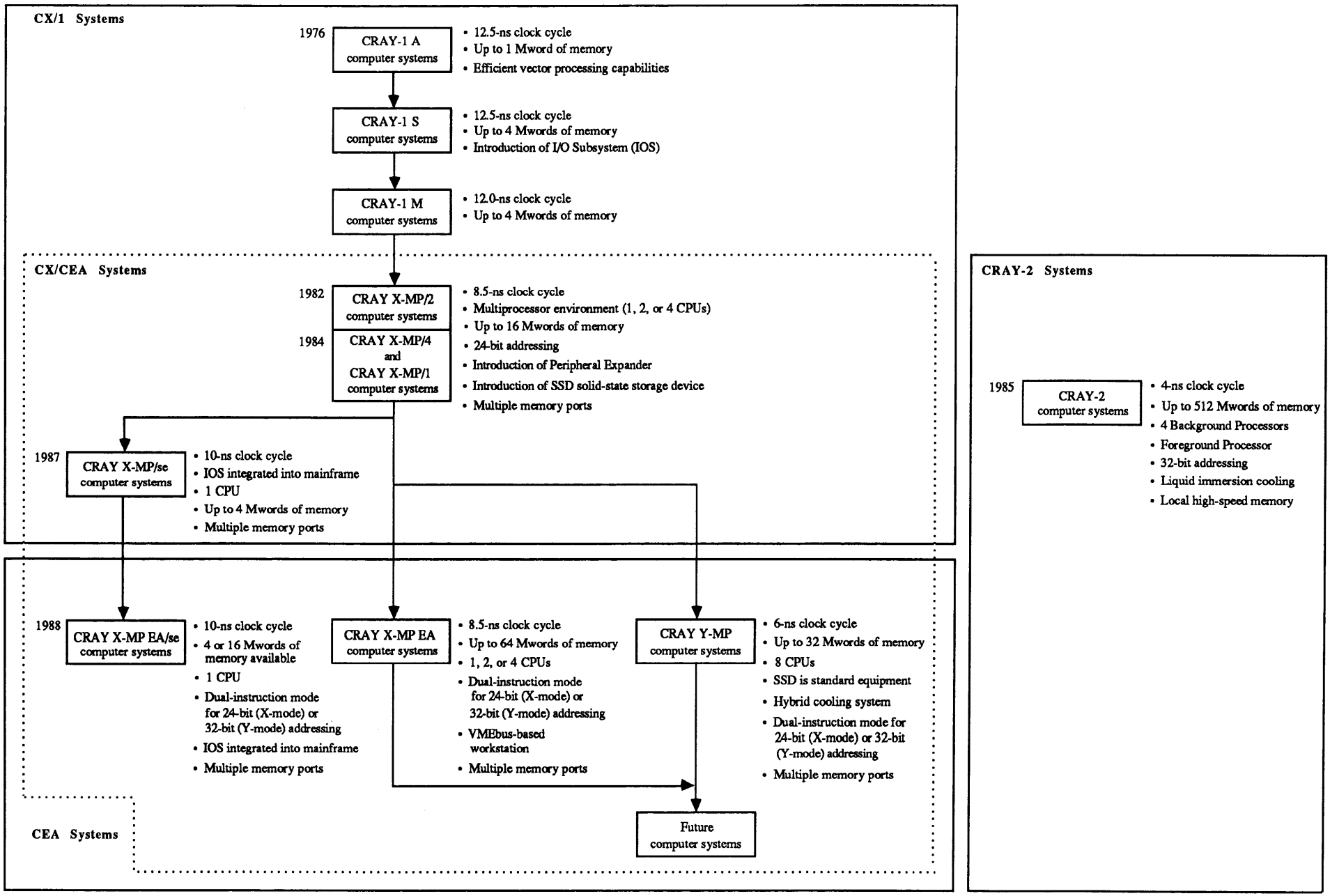
- Send a facsimile of your comments to the attention of "Software Information Services" at fax number (612) 683–5599.
- Use the postage-paid Reader's Comment form at the back of this manual.

- Write to us at the following address:

  Cray Research, Inc.
  Software Information Services Department
  655F Lone Oak Drive
  Eagan, MN 55121

We value your comments and will respond to them promptly.

Hardware Product Line



CRAY RESEARCH, INC.

The following list defines architecture terms:

Term	Definition
CX/1 systems	This group includes all models of the CRAY X-MP and CRAY-1 computer systems. It is characterized by 24-bit addressing capabilities.
CEA systems	This group includes all models of the Extended Architecture (EA) series, which are the CRAY Y-MP and CRAY X-MP EA computer systems. It is characterized by 32-bit addressing capabilities.
CRAY-2 systems	This group includes all models of the CRAY-2 computer systems. It is characterized by 32-bit addressing capabilities, large common memories, and immersion cooling.
CX/CEA systems	This group designates all models of CRAY X-MP computer systems plus all models of the CRAY Y-MP and CRAY X-MP EA computer systems. It does not include CRAY-1 computer systems.
EAM bit (hardware)	In CX/1 systems, the EAM bit is the Enhanced Addressing Mode bit in the Flag register. When set, it sign-extends certain instructions for memory addressing in 8- and 16-Mword systems. In CEA systems, the EAM bit is the Extended Addressing Mode bit in the Flag register. It is set by the operating system to select either 24- or 32-bit addressing.
EMA feature (software)	In CX/1 systems, EMA is the Extended Memory Addressing feature for 8- or 16-Mword systems.
X-mode	This term refers to the 24-bit addressing mode in CEA systems. The operating systems select this mode with the EAM bit in the Exchange Package.
Y-mode	This term refers to the 32-bit addressing mode in CEA systems. The operating systems select this mode with the EAM bit in the Exchange Package.

Contents

Page

Figures

Tables

Introduction to Vectorization [1]

Figures

Tables

Introduction to Vectorization [1]

Vectorization is a form of parallel processing in which array elements are processed by groups. For many applications, vectorization is the most important optimization feature of a Cray Research system. A loop that is executed with vectorized code typically runs 10 times faster than when executed with conventional (scalar) code.

The CF77 compiling system vectorizes Fortran code when it determines that this will not affect the program results. An understanding of vectorization will help you get the best performance. Of the components of the compiling system, the two that are concerned with vectorization are the FPP dependency analyzer and the CFT77 compiler. FPP makes modifications of the actual source code to allow vectorization, while the compiler modifies the *intermediate text*, a representation of the program at a later stage of processing.

Illustrating vectorization

1.1

This subsection describes vectorization for those who are unfamiliar with it. A more technical discussion is given in later subsections.

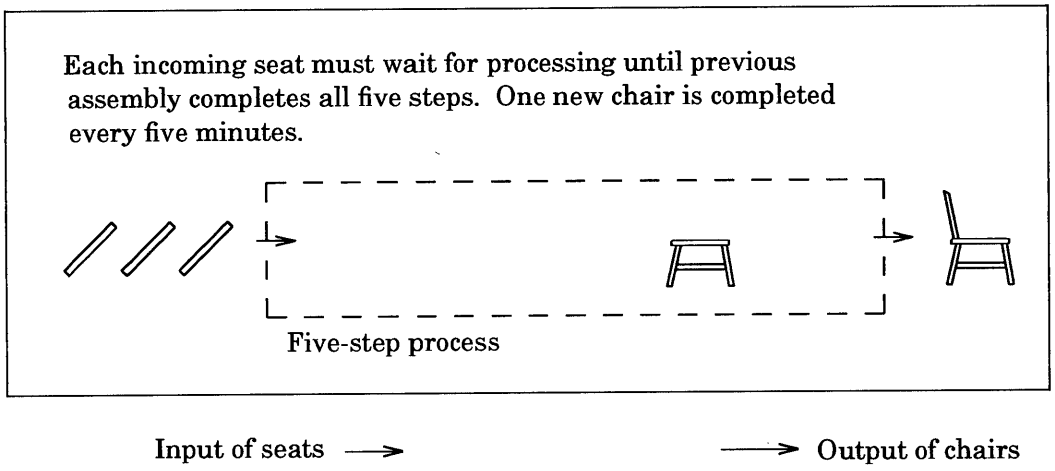
A *scalar* is a single value. *Scalar processing* consists of performing logical and arithmetic operations on scalars one at a time.

A *vector* is a series of values or *vector elements*; for example, a vector can be derived from a Fortran array or part of an array, though an array does not have to be processed as a vector. A hardware operation for processing values, such as an add operation, consists of multiple steps. *Vector processing* or *vectorization* allows these internal steps to process vector elements in close sequence, so that a vectorized operation can simultaneously process as many elements as the number of steps in the operation.

Vectorization works something like an assembly line in manufacturing. To illustrate, Figure 1 compares two ways to carry out a five-step process for making chairs, in which each step takes one minute:

- In one process, one assembly must go through all five steps before the next assembly can begin the first step. One chair is produced every five minutes.
- With the use of an assembly line, each step in the sequence is performed as soon as the previous step is complete. In this way, the first chair is completed after five minutes, and one new chair is completed each minute after that.

Scalar processing: one at a time



Vector processing: assembly line

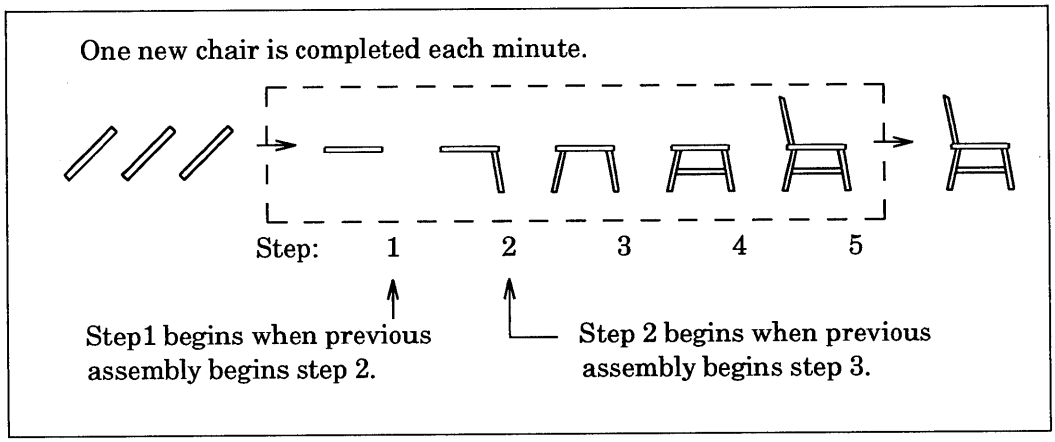


Figure 1. Scalar versus vector, illustrated

Processing an array
1.1.1

Like manufacturing a chair, processing an array in a Fortran loop is a multi-step process that can be performed one step at a time, or by a process resembling an assembly line.

To process an array in either vector or scalar mode a loop must do the following:

- Load elements from memory to a register
- Process the elements, placing the results in another register
- Store the results back to memory

Conventional (scalar) code is a one-at-a-time process: each element must finish the final step of processing before the next element can begin the first step; that is, each loop iteration begins when the previous iteration ends. But with vector code, as in an assembly line, each element begins the first step when the previous element finishes only the first step, rather than the final step.

Vector hardware
1.2

Vector processing is made possible by the following hardware features, among others:

- *Pipelining* within the functional unit for an operation allows each step of the operation to pass its result to the next step after only one clock period (CP); for example, an add operation has seven steps taking seven clock periods. Pipelining is shown schematically in Figure 2.
- *Chaining* (not used in CRAY-2 systems) allows the movement of elements to continue from one vector operation to another, so that a process including more than one vector operation is executed as one long vectorized operation.
- *Vector registers* are registers that hold up to 64 elements.

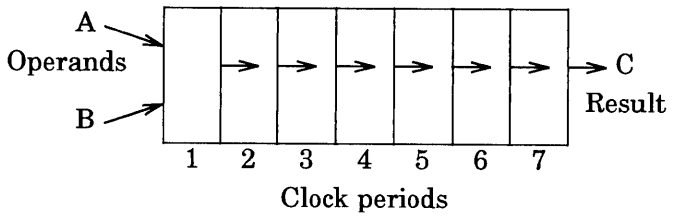


Figure 2. Pipelining in a functional unit

Vector operations 1.2.1

An operation that is pipelined, as described in the previous paragraph, is a *vector operation*. Vector operations include those to load, store, add, subtract, and multiply vectors. These vector operations are referred to as *vector load*, and so forth.

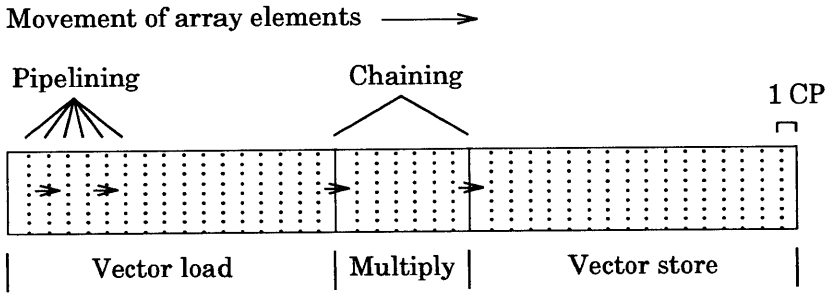
Example:

```
      DO 10, I=1,1000
        A(I) = B(I) * C(I)
10 CONTINUE
```

The operations generated by this code, executed on a CRAY Y-MP system, are timed as follows:

Operation	Time in clock periods
Vector load	17
Vector multiply	7
Vector store	17

Therefore, for a loop that multiplies two vectors, element by element, the length of the load-multiply-store "assembly line" is 17+7+17=41 steps. (The two vector operands are loaded simultaneously, excluding instruction issue time.) When these operations are vectorized and chained, the first result is stored after 41 CPs; after that, one new result is stored on each clock period. This operation is shown schematically in Figure 3.



For illustration only. Functional units are not physically arranged as shown.

Figure 3. Pipelining and chaining

Scalar and vector processing
1.2.2

The following loop adds each element of array KK with the corresponding element of array LL and stores the results in array JJ.

```
      DO 10 I=1,3
         JJ(I) = KK(I) + LL(I)
   10 CONTINUE
```

With scalar processing, this loop performs the following operations (if it is not unrolled):

- Read one element of Fortran array KK.
- Read one element of LL.
- Add the elements.
- Write the result to the Fortran array JJ.
- Increment the loop index by 1.
- Repeat the above sequence for each succeeding array element until the loop index equals its limit.

With vector processing, the preceding loop performs the following vector operations:

- Load a series of elements from array KK to a vector register and a series of elements from array LL to another vector register (these operations occur simultaneously except for instruction issue time).
- Add the corresponding elements from the two vector registers and send the results to another vector register, representing array JJ.
- Store the register used for array JJ to memory.
- This sequence would be repeated if the array had more elements than the maximum elements used in vector processing.

Changes in the order of processing
1.3

Inherent to vector processing is a change in the order of operations performed on individual array elements, for any loop that includes two separate vectorized operations. For example, the following loop performs two separate additions on arrays.

```
      DO 10 I=1,3
         L(I) = J(I) + K(I)
         N(I) = L(I) + M(I)
   10 CONTINUE
```

With scalar processing, the two statements within this loop are each executed three times, with the two operations alternating: `L(I)` is calculated before `N(I)` in each iteration. The new value of `L(I)` is used to calculate the value of `N(I)`. This order of operations is shown in Table 1.

Table 1. Scalar processing order and results

Event	Operation	Values
1	L(1)=J(1)+K(1)	7 = 2 + 5
2	N(1)=L(1)+M(1)	11 = 7 + 4
3	L(2)=J(2)+K(2)	–1 = (–4) + 3
4	N(2)=L(2)+M(2)	5 = (–1) + 6
5	L(3)=J(3)+K(3)	15 = 7 + 8
6	N(3)=L(3)+M(3)	13 = 15 + (–2)

With vector processing, however, the first line within the loop processes all elements of the array before the second line is executed. The order of operations performed on individual array elements is shown in Table 2. Notice that this order differs from that shown for scalar processing in Table 1.

Table 2. Vector processing order and results

Event	Operation	Values
1	L(1)=J(1)+K(1)	7 = 2 + 5
2	L(2)=J(2)+K(2)	–1 = (–4) + 3
3	L(3)=J(3)+K(3)	15 = 7 + 8
4	N(1)=L(1)+M(1)	11 = 7 + 4
5	N(2)=L(2)+M(2)	5 = (–1) + 6
6	N(3)=L(3)+M(3)	13 = 15 + (–2)

As shown in Tables 1 and 2, results for each array element are equivalent in scalar and vector processing. This equivalence is a fundamental requirement of vector processing.

In the preceding code example, the values calculated on the first line of the loop are used in the operation in the second line. The later use of a result within this loop does not change the results when the order of operations is changed. Within other loops, however, later use of a calculation can cause different final results and will therefore inhibit vectorization; this is referred to as a *data dependency*. See the section beginning on page 39.

CPU Vector Architecture [2]

CPU Vector Architecture [2]

This section summarizes the system architecture of CRAY Y-MP and CRAY-2 systems, emphasizing aspects that affect vectorization. To display the model and configuration of the system you use, enter the `target` command.

CRAY Y-MP systems

2.1

CRAY Y-MP systems offer a wide range of computational power and memory capacity with a high degree of field upgradability. Figure 4, page 11, shows the block diagram of a CRAY Y-MP system.

The CRAY Y-MP product line includes three basic configurations: CRAY Y-MP8, CRAY Y-MP4, and CRAY Y-MP2 systems. These systems can be configured with up to 8, 4, and 2 central processing units (CPUs), respectively.

All currently manufactured CRAY Y-MP systems have a clock period of 6.0 nanoseconds.

Computer resources

2.1.1

The CPUs of a CRAY Y-MP system make use of these areas of the computer:

- Central memory can be 16 to 128 Mwords. (A Cray word consists of 64 bits.) Memory is shared by all of the CPUs and the I/O section. Each CPU in the system has four parallel memory ports. Each port performs specific functions, allowing different types of memory transfers to occur simultaneously. Special hardware minimizes delays caused by memory conflicts.
- The I/O section, shared by the CPUs, consists of channels that communicate with the I/O subsystem (IOS) and the optional SSD solid-state storage device.

- The interprocessor communication section allows two or more CPUs to work simultaneously on the same program. Communication and synchronization between CPUs are made possible by shared registers in this section.
- All CPUs in a system share a single real-time clock (RTC), consisting of a 64-bit counter that is incremented on each clock period. Because the clock advances synchronously with program execution, it can time the program to an exact number of clock periods.

CPU computational section 2.1.2

Each CPU has its own computational section consisting of registers and functional units.

The *primary* registers can be accessed directly by central memory and by the functional units. These are the address (A), scalar (S), and vector (V) registers. Scalar (S) and vector (V) registers serve as the source and destination for arithmetic and logical instructions. The A and S registers are supported by the B and T registers, respectively, which are called *intermediate* and cannot be accessed by the functional units.

Functional units implement algorithms or portions of the instruction set, such as add or multiply. Each functional unit can be accessed by one group of registers (address, scalar, or vector); the floating-point functional unit can be accessed by either the scalar or the vector registers. The functional units are as follows:

- Vector functional units are Vector Add, Vector Shift, Full Vector Logical, Second Vector Logical, and Vector Population/Parity.
- Scalar functional units are Scalar Add, Scalar Shift, Scalar Logical, and Scalar Population/Parity/Leading Zero.
- Floating-point functional units are Floating-point Add, Floating-point Multiply, and Reciprocal Approximation. The operands to the floating-point functional units can be either a pair of scalar registers or a pair of vector registers.
- Address functional units are Address Add and Address Multiply.

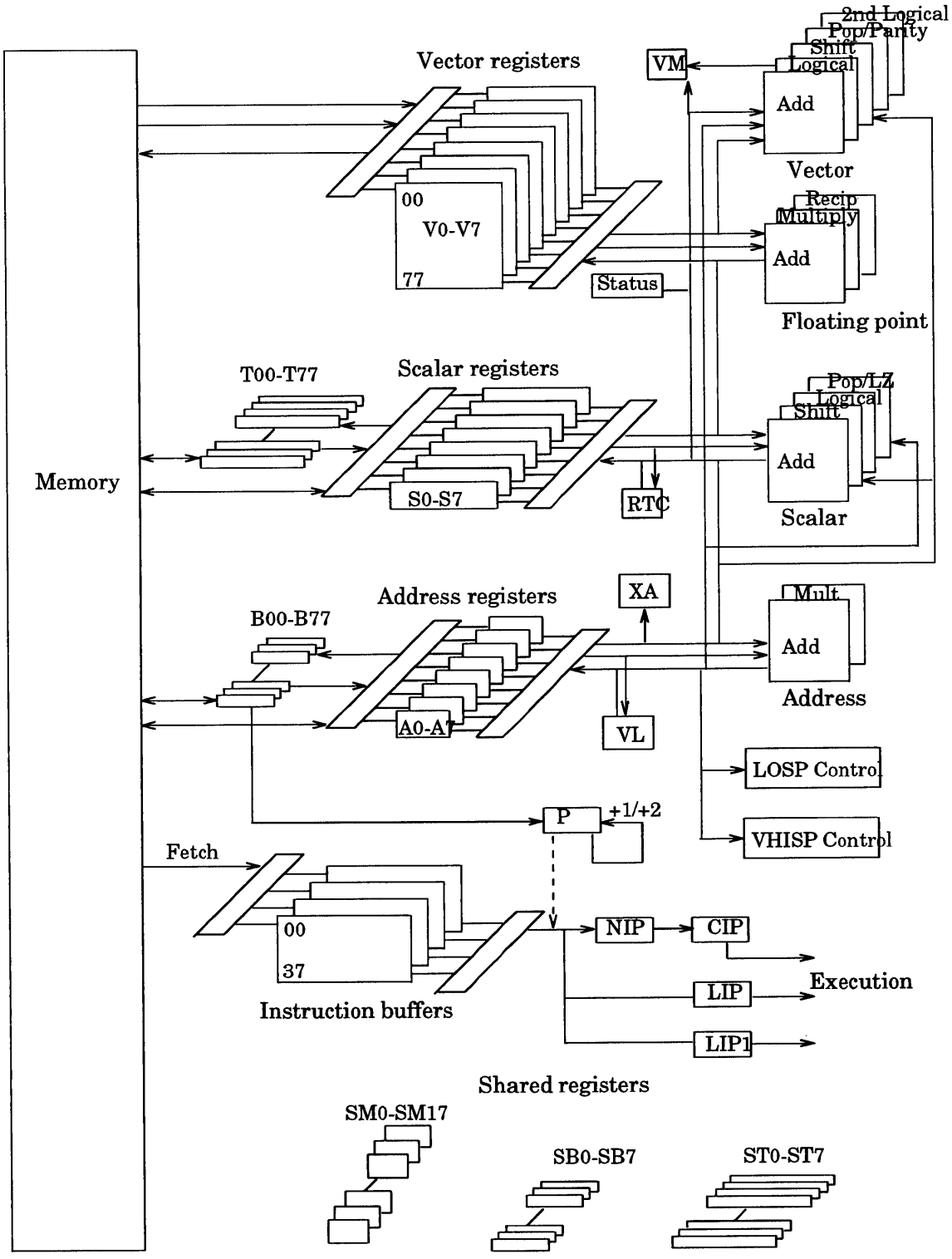


Figure 4. CRAY Y-MP block diagram

Vector registers
2.1.2.1

Each CPU contains eight vector (V) registers. Each V register consists of 64 elements, each containing 64 bits. A single vector instruction causes successive elements from a V register to enter a functional unit in successive clock periods. Vector processing allows a set of operands to be used to compute a series of results.

The vector length (VL) register specifies the number of elements in the vector register to be processed by a vector instruction. The contents of the VL register range from 1 to 64_{10}.

The 64-bit vector mask (VM) register represents a selection of vector elements stored in a V register. Each bit in the VM register corresponds to one element (a one-word value) in the vector. The mask uses vector merge and test instructions to perform operations on individual elements.

CRAY Y-MP architecture and vector processing
2.1.3

Segmented functional units and chaining are two major architectural features that have a great impact on CRAY Y-MP vector processing.

Segmented functional units
2.1.3.1

Each CRAY Y-MP CPU contains several functional units, each designed for a special purpose. Because these functional units operate independently of each other, operations (such as adds and multiplies) can occur in parallel. Additionally, the functional units are fully segmented. That is, the intermediate steps required to complete an operation are broken down into one-clock-period segments. Results occur at the rate of one per clock period. Segmenting is the basis of pipelining, as described and illustrated on page 3.

Depending on their complexity, different functional units have different numbers of segments. For example, the reciprocal approximation functional unit has more segments than the floating-point multiply functional unit. The number of segments determines the number of clock periods required to compute the result for a scalar or the first result for a vector. Subsequent vector results occur at the rate of one per clock period.

Chaining
2.1.3.2

The functional units in each CRAY Y-MP CPU operate independently (in parallel). This capability allows the output from one operation to directly become the input to another functional unit. Chaining occurs when the resultant vector register becomes an operand vector register for a second

calculation. As soon as the first result arrives, the second calculation begins, rather than waiting for the entire first vector operation to finish. For example, when the first result from a vector multiply is available, it can be used as input to the vector add function.

Consider the following loop:

```
      DO 8 I = 1, N
        X(I)   = A(I) * B(I) + C
    8 CONTINUE
```

The above code would generate code that would be chained in the following way. Each numbered item shows operations that occur simultaneously except for instruction issue time:

1. Load scalar `C` from memory into the S register. (This can occur at various times but typically occurs first.)
2. Load arrays `A` and `B` from memory into vector registers V0 and V1.
3. Multiply V0 by V1 in functional unit, storing result to register V2.
4. Add scalar `C` to elements from V2, storing result to register V3.
5. Store V3 to memory.

Chains of three or more functional units are possible but unusual. Although chains of two functional units are typically the maximum for CF77-generated code, the two data paths from memory (LOAD paths) and to memory (STORE path) can also be considered functional units. Therefore, optimizing loads and stores to and from memory also promotes chaining on CRAY Y-MP systems.

CRAY-2 systems
2.2

Figure 5 shows a block diagram of a CRAY-2 system. The system contains the following components:

- A single foreground processor (FGP) that controls and monitors system operation
- Common memory consisting of 256 or 512 million words of dynamic memory, or 64 or 128 million words of static memory.

Foreground processor
2.2.1

A single foreground processor is the heart of the CRAY-2 system. It controls and monitors system operation by responding to background processor requests and sequencing channel communication signals. The primary function of the foreground processor is real-time response to requests from a variety of sources (called channel nodes). The foreground processor polls the various nodes in the system and takes action when necessary.

To communicate with the other system resources, such as CPUs or disks, the foreground processor uses a series of four *channel loops*. A channel loop consists of various controllers and ports. Each controller or port is called a *channel node*. A channel loop must contain the following nodes: common memory port and background processor port.

Background processor
2.2.2

A CRAY-2 system contains two or four identical background processors. Each background processor includes the components described in the following subsections.

Computational section
2.2.2.1

A background processor's computational section contains the operating registers and functional units necessary to perform arithmetic and logical calculations.

There are three types of operating registers: address (A), scalar (S) and vector (V) registers.

There are four kinds of functional units: address, scalar, vector, and floating-point. Functional units receive operands from registers, perform an operation on the operands, and return the result to registers. Functional units can operate simultaneously. Each functional unit can deliver one result per clock period.

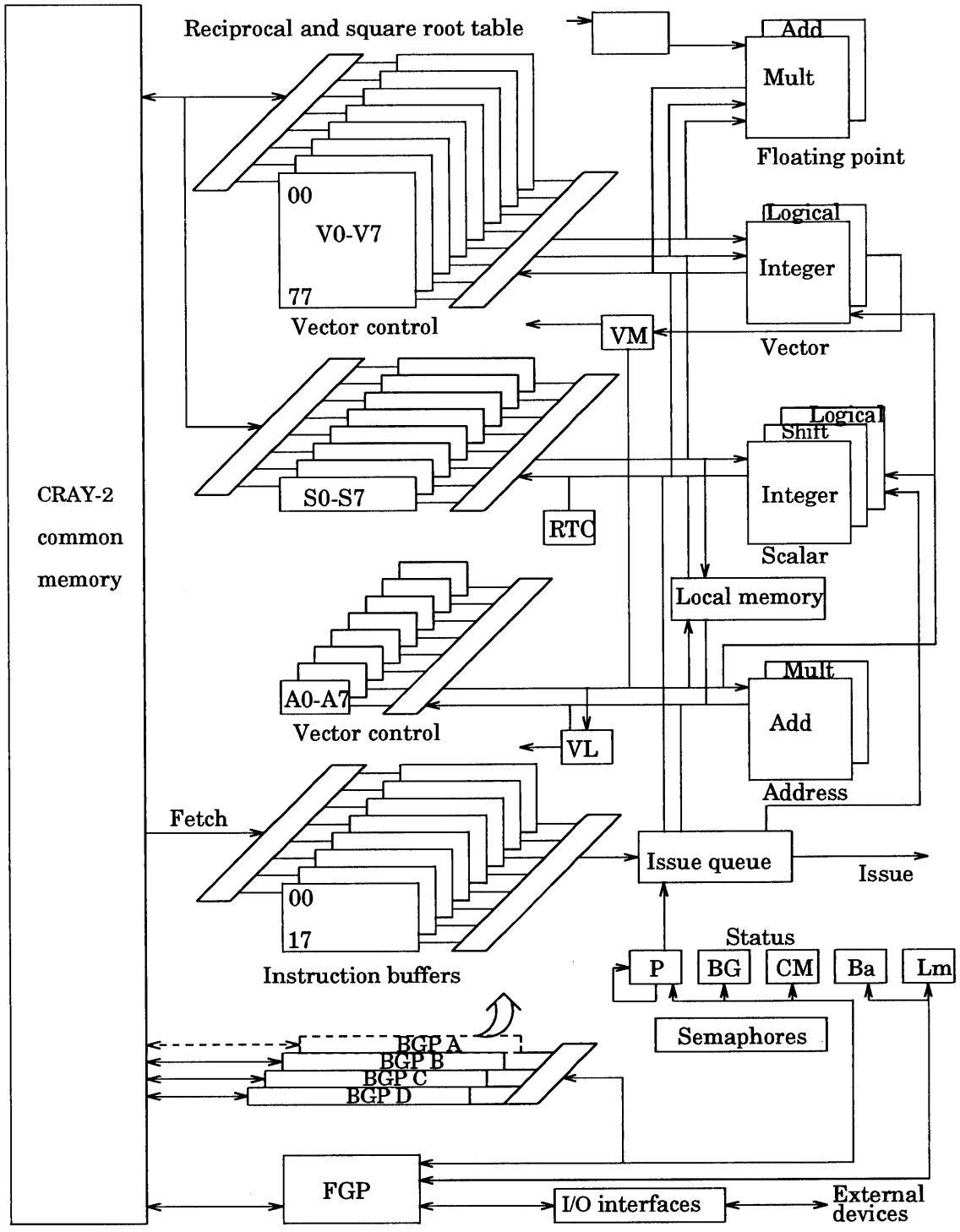


Figure 5. CRAY-2 system block diagram

The functional units are as follows:

- Address functional units are Address Add and Address Multiply.
- Scalar functional units are Scalar Integer, Scalar Shift, and Scalar Logical.
- Vector functional units are Vector Integer and Vector Logical.
- Floating-point functional units can operate on either a pair of scalar registers or a pair of vector registers. The units are Floating-point Add and Floating-point multiply.

Control section
2.2.2.2

Each background processor contains an identical, independent control section. Each background processor's control section coordinates the operations of the operating registers and functional units, along with the other functions of the background processor.

The control section includes a program address (P) register indicating the address of the current instruction parcel; eight independent instruction buffers that allow program loops to execute without additional common memory references; and an instruction issue mechanism that issues one instruction every two clock periods.

Local memory
2.2.2.3

Each background processor contains 16K 64-bit words of local memory. This memory holds scalar operands during a computation period and then returns the data to common memory. The local memory is also used for temporary storage of vector elements when these elements are used more than once in a computation in the V registers.

CRAY-2 architecture and vector processing
2.2.3

CRAY-2 system architecture promotes Fortran code efficiency. At the same time, loop vectorization can exploit system efficiency. Knowing how vectors are handled by CRAY-2 processors helps you code for improved efficiency. Two major architectural features that have a great impact on CRAY-2 vector processing are segmented functional units and functional unit overlap.

Segmented functional units
2.2.3.1

A CRAY-2 background processor contains a number of functional units, each designed for a special purpose. Because these functional units operate independently of each other, operations (such as adds and multiplies) can occur in parallel. Additionally, the functional units in a CRAY-2 system are fully segmented. That is, the intermediate steps required to complete an operation are broken down into one-clock-period segments. This allows results to be generated at the rate of one per clock period.

Depending on their complexity, different functional units will have different numbers of segments. For example, the reciprocal approximation functional unit has more segments than the floating-point multiply functional unit. The number of segments determines the number of clock periods to compute the result for a scalar operation or the first result for a vector operation. Subsequent vector results are available at the rate of one per clock period.

Functional unit overlap
2.2.3.2

The functional units in the CRAY-2 system operate independently (in parallel). This independence allows two vector operations to be running simultaneously.

All of the functional units on a CRAY-2 system are independent of each other and therefore can be combined in parallel operations. Combinations of three or more functional units are not possible, however, due to limitations on the number of data paths.

Parallel usage of two functional units in vector operations is rather uncommon in CF77-generated code. However, the single path that moves data from memory and that stores data to memory can be considered a functional unit. Hence, optimizing loads and stores from and to memory also promotes speed on a CRAY-2 system.

Essentials of Vectorization [3]

Section 1 introduced the underlying concepts of vectorization. This section introduces the basic elements of accomplishing vectorization on a practical level.

Although the CF77 compiling system performs extensive vectorization automatically, a knowledge of vectorization will help you increase the degree of vectorization, as well as removing constructs that create unnecessary work for your program. This section describes the essentials of what you need to know, such as the kinds of expressions or code constructs that can be vectorized by the compiling system.

Note

Examples in this section include comments such as "Vectorization inhibited" and "Inhibits vectorization." If such a comment appears on a line within a loop, it applies to that line in particular. If the comment appears on the loop's DO statement, it applies to the loop as a whole, or the interaction of two or more statements within the loop.

Terminology

3.1

This subsection defines terms and concepts used in this manual.

- A *vector* is a series of values on which instructions operate: this can be an array or any subset of an array (such as a row, column, or diagonal) in which the intervals between the array elements' memory locations are constant. When arithmetic or logical operations are applied to vectors, it is referred to as *vector processing*.
- *Vector length* is the number of elements in a vector; maximum 64. This means that an array of 640 elements must be divided into at least ten vectors. With ten vectors, the length of each would be 64; other combinations could be 20 vectors of length 32, ten vectors of length 60 with one of length 40, and so on. Each vector must be separately loaded, processed, and stored.

- The *stride* is the interval between memory locations for successive elements of a vector. A *constant stride* is an interval that is the same for all consecutive elements of a vector. Vectorization requires a constant stride, meaning that an array is processed in sequences such as `A(1), A(2), A(3), ...` , or `B(2), B(4), B(6), ...` A stride that is not constant is illustrated by a sequence such as `A(1), A(2), A(3), A(5), A(8), A(13).`

- A *dependency* is anything that causes vector and scalar code to give different results. A *recurrence* or *data dependency* is an expression, within a loop, that requires a value calculated in a previous iteration of the loop in order to be evaluated. This situation is also referred to as a *vector dependency*. See sections beginning on pages 39 and 39.

- A *chime* is a sequence of vector operations that can be chained with a single vector load and store. The limitation on such a sequence is that the same vector functional unit cannot be used twice in the same chain. Therefore, a loop that contains two vector adds, for example, contains at least two chimes and the overhead of at least two load/stores, because there is only one Vector Add functional unit.

Branches and blocks
3.1.1

A *branch* is a transfer of control; that is, any conditional code. Branches result from `GOTO`, arithmetic `IF`, or alternate return. For example, the line `IF (N.EQ.0) M = 100` is considered a branch because the assignment to `M` is conditional.

A *block* is a section of code with no explicit or implicit branches, including simple cases such as the `IF` statement mentioned in the preceding paragraph.

A *backward branch* is a transfer of control to a preceding point in the program. A *forward branch* goes to a later point in the program.

Loop invariant
3.1.2

A *loop invariant* is a constant or simple variable that is referenced but not redefined in the course of a loop. An array element is also considered to be a loop invariant if the subscript expression(s) is/are loop invariant. In the following code, `M`, `N`, `X`, `I`, `B(I)`, and `4` are examples of loop invariants, within the innermost loop, `DO 10`:

```
      DIMENSION A(200), B(200), C(200), D(200)
      PARAMETER (X=101, M=100, N=100)
      K=0
      DO 20 I = 1, M
        DO 10 J = 1, N
          K= K + X + I
          L = J + 4 + M
          A(J) = B(I) * C(L) + D(K)
10      CONTINUE
20    CONTINUE
```

Scalar temporary 3.1.3

A *scalar temporary* is a simple variable defined and later referenced during each pass through a loop, but is not referenced outside the loop. The compiler can replace a scalar temporary with a temporary vector or eliminate it.

In the following code segment, the Y variable is a scalar temporary:

```
      DIMENSION A(100), R(100)
      DIMENSION S(100,100)
      DO 20 J = 1, 100
        Y = R(J) * S(1,J)
        A(J) = Y
20    CONTINUE
      END
```

Loop counter 3.1.4

A *loop counter* is an integer variable that is incremented or decremented by an integer constant expression on each pass through the loop. Additional conditions for determining whether an element is a loop counter are as follows:

- The only operators allowed in the expression defining the variable are addition and subtraction; the expression cannot include multiplication, division, and exponentiation.
- If the variable defines itself in the course of the loop, and the result could alternate in sign (for example, `I = 1 - I`), the variable is still considered a loop counter, but it will inhibit vectorization.
- If a variable is defined as the sum or difference of the same variable (for example, `I = I + I`), the variable will inhibit vectorization.

A loop's index is one kind of loop counter.

In the following Fortran sequences, I, J, L, JJ, and N are loop counters. Of these, only JJ inhibits vectorization. JJ is defined first by JJ = JJ + 1; a subsequent statement (JJ = 1 - JJ) changes its sign. This means that JJ alternates in sign on each pass through the loop, since it is given a nonzero initial value.

```
      DIMENSION A(200), B(200)
      K = 1
      J = 10
      M = 100
      DO 10 I = 10, 20
        J = J + 75
        L = K - J
        N = L + M
        B(I) = A(N + I)
        JJ = JJ + 1
        JJ = 1 - JJ
   10 CONTINUE
      END
```

A loop counter may be incremented or decremented more than once if the effect of the two assignments is the same as a single assignment. In the following code, the two assignments to the J variable are equivalent to J=J-1 for vectorization purposes:

```
      DIMENSION A(200), B(200)
      DIMENSION C(200)
      J = 0
      DO 10 I=10,20
        J = J + 1
        A(J) = B(J) * C(J)
        J = J - 2
   10 CONTINUE
      END
```

Vector array reference

3.1.5

A *vector array reference* is an array element reference whose subscript expression is not a loop invariant. It is an array that is processed in vector registers.

The following code shows examples of vector array references `X(J)`, `L(I+2)`, `TAB(I, N+3)`, and `SQ(I, 2*J+4)`.

```
      DIMENSION X(200), TAB(200,100)
      DIMENSION L(200), SQ(200,100)
      N = 5
      DO 10 I = 1, 10
        J = I + 4
        X(J) = L(I+2)
        TAB(I, N+3) = SQ(I, 2*J + 4)
   10 CONTINUE
      END
```

Vectorizable expression
3.1.6

A *vectorizable expression* is an arithmetic or logical expression that consists of a combination of any of the following:

- Loop invariants
- Loop counters
- Vector array references
- Scalar temporaries
- A function with a vector version that has any of the preceding as arguments. This includes most CF77 intrinsic functions and math and scientific library routines.

A *vectorizable loop* is an innermost loop that contains only vectorizable expressions (that is, expressions for which the CF77 compiling system can produce vector code).

Character expressions are not vectorized.

General requirements for vectorization
3.2

`DO` loops, `DO WHILE` loops, and `IF` loops can be vectorized if they meet certain requirements. These can be summarized as follows:

- In a nested structure, only innermost loops can be vectorized.
- Vector and scalar versions of a given code must produce equivalent results.

- Other requirements are based on hardware considerations and limits on the complexity of a loop.

Innermost loops 3.2.1

Within a group of nested loops, only innermost loops are vectorized. Example:

```
      DO 10 I = 1,N      ! Vectorization inhibited
        A(I) = 0
        DO 20 J = 1,M    ! Vectorization permitted
          B(J,I) = 0
20    CONTINUE
10  CONTINUE
```

If you manually split the outer loop in the preceding example (`DO 10 I=`) so that the assignments are performed in separate loops, both loops can vectorize, as in the following:

```
      DO 10 I = 1,N      ! Vectorization permitted
        A(I) = 0
10  CONTINUE
      DO 20 I = 1,N      ! Vectorization inhibited
        DO 30 J = 1,M    ! Vectorization permitted
          B(J,I) = 0
30    CONTINUE
20  CONTINUE
```

A loop does not have to be a `DO` loop to be a candidate for vectorization. A `GOTO` structure is a potentially vectorizable loop if it generates, in intermediate text, constructs similar to those generated by `DO` loops. The following is an example of a vectorizing loop that is not a `DO` loop:

```
1.      SUBROUTINE VECTOR_IF_LOOP(A,B,C)
2.      REAL A(100), B(100), C(100)
3.      I = 1
4.   10 CONTINUE
5.        A(I) = B(I) * C(I)
6.        I = I + 1
7.      IF(I .LE. 100) GO TO 10
8.      END

*** *** FF8004 [vector ]
        Loop starting at line 4 was vectorized.
```

If a loop cannot be vectorized in its current form, you may be able to modify it so that the compiler can vectorize it. Some small innermost loops are candidates for *unwinding*. Unwinding consists of making several copies of the body of the loop, so that the loop is converted to straight-line code. This makes the next outer loop the innermost nested loop and CF77 attempts to vectorize it. If you compile with the Loopmark option (using `cf77 -Wf"-em"`), an unwound loop is indicated by a `W` notation to the left of the code. Unwinding is discussed more fully in *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071.

Numerical differences 3.2.2

To increase the opportunities for vectorization, the compiler weakens the requirement for vector-scalar equivalence by ignoring two sources of potential numerical differences:

- Computer floating-point arithmetic is not associative. For example, a scalar loop summing the elements of an array is transformed into a vector reduction, causing the elements to be summed in a different order. You must identify the cases in which this leads to an unacceptable difference in the sum.
- Some interrupts may be handled at different points in the scalar and vector versions of the program. Such interrupts can arise from operand range errors and floating-point exceptions.

Subprogram references 3.2.3

Loops containing calls to function subprograms and subroutines do not vectorize, with these exceptions:

- Subprogram calls that are expanded inline.
- Functions with vector versions, identified by the `VFUNCTION` directive; see the following subsection.
- Certain intrinsic and library functions (including `MOD`, `ABS`, and `RANF`). Examples:

```
CALL BBB(B,N)          ! Inhibits vectorization
A(I) = MYFUNC(B(I))    ! Inhibits vectorization
A(I) = SIN(B(I))       ! Permits vectorization
```

Full vectorization
3.2.4

The CF77 compiling system attempts to generate machine instructions that will execute a particular code construct as efficiently as possible. In many cases, the operations within a Fortran looping construct can be implemented entirely with vector instructions. This is called *full vectorization*.

A loop is a candidate for full vectorization when all of the following statements are true:

- There is at least one vector array reference on the left side of the equal sign.
- All the simple variables in the loop are loop counters without possible alternating signs, loop invariants, or scalar temporaries.
- The only kinds of terms to be defined (that is, appearing on the left of an equal sign) are vector array references, scalar temporaries, or loop induction variables.
- Any functions used in the loop are statement functions, vectorizable intrinsic functions with vectorizable expressions as arguments, or vectorizable CAL functions specified in `VFUNCTION` directives.

Conditions that inhibit vectorization
3.3

Vector code cannot result if any of the following are in an innermost loop:

- Any reference to external code that cannot be vectorized. This includes:
  - I/O statements (these generate library calls). But note that an implied-`DO` list in an I/O statement does vectorize.
  - References to a function that does not have a vector version.
  - A reference to an external function or subroutine that is not expanded inline.
  - Any `RETURN`, `STOP`, or `PAUSE` statement; these generate library calls. However, a jump from within a loop to one of these statements does not inhibit vectorization.
- Obsolete conditional statements: three-branch `IF`, assigned `GOTO`, and computed `GOTO`. These cannot be converted to vector hardware instructions.

- Backward branches other than the one that forms the loop.
- The presence of source directives NOVECTOR, NEXTSCALAR, or SUPPRESS. Vectorization can also be disabled from the `cf77` command line with `-Wf"-o"` options `off`, `noscalar`, or `novector`.
- A statement branch into the loop from outside the loop. Such code violates the ANSI standard but is permitted by the CF77 compiling system.
- Array bounds checking.
- Dependencies (constructs that produce different results in scalar and vector mode):
  - A recurrence, with exceptions shown in the section beginning on page 39.
  - Ambiguous subscript references. If the ambiguity can be limited during compilation, the loop can be conditionally vectorized.

Complexity 3.3.1

Generally, complexity can prevent vectorization, because the required analysis is judged to be too demanding of system resources relative to the performance improvement that is anticipated in the generated code.

Note this tradeoff: smaller loops are generally more likely to vectorize, but larger loops have less overhead per operation and give better performance if they vectorize.

The `cf77 -Wf"-o aggress"` option raises certain internal limits used by the compiler, allowing loops of greater complexity to be vectorized. This option does not increase dependency analysis and may increase compile time. See *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071.

Memory contention

3.4

The performance improvement normally expected with vectorization can be degraded by *memory contention*, which is the conflict between successive accesses to the same memory bank. Each access to a location in a memory bank causes that bank to be unavailable for some number of clock periods. Any attempts to access a bank that is currently unavailable are held until the bank is free.

Consecutive Fortran array elements are stored in distinct memory banks, which are numbered consecutively. For example, given an array declared `REALA(100)`, element `A(1)` is stored in bank *n*, `A(2)` in bank *n*+1, and so on. Memory addresses for elements in a two-dimension array are in the order corresponding to elements moving down a column of the array; therefore, a single column is stored in different banks, and a row is stored all in one bank. (Recall that a column contains elements whose first subscript is incremented; a row contains elements whose second subscript is incremented while the first is constant.)

Memory conflicts, then, are caused by successive accesses to the same memory bank. The following example illustrates one frequent cause of bank conflicts: the accessing of elements consecutively within a row.

```
REAL A(256,100), B(256,100)
...
DO I=1,256
  DO J=1,100                ! Accesses rows
    A(I,J) = B(I,J) * 2
  END DO
END DO
```

In this example, the order of access is: `A(1,1)`, `A(1,2)`, `A(1,3)`, ...; these are consecutive elements in a row. As just explained, a row is stored in one memory bank, so each successive access is held (delayed). The performance of the vector load and store is degraded depending on the bank-busy time for your system's memory. Typically, a vector load or store without memory contention runs 5 to 8 times faster than the same vector load or store with the greatest memory contention.

Enter the `target` command to see how many memory banks are configured on your system, and the length (in clock periods) of a memory hold on your system, indicated by the heading `bankbusy=`.

Memory optimization 3.4.1

The best way to ensure efficient memory access is to write loops so that elements are accessed down columns (incrementing the first subscript on consecutive elements) and vectors to be loaded or stored have odd strides. An odd stride guarantees that successive accesses are not to the same memory bank.

Following are two ways to modify the preceding example so that the resulting vector load and store use an odd stride:

Original:

```
REAL A(256,100), B(256,100)
...
DO I=1,256
   DO J=1,100
      A(I,J) = B(I,J) * 2
   END DO
END DO
```

Modified:

```
REAL A(256,100), B(256,100)
...
DO J=1,100        ! "Flipped" DO statements
  DO I=1,256
    A(I,J) = B(I,J) * 2
  END DO
END DO
```

The access in the modified code is now down the columns (stride 1) whereas the access in the original code is along the rows (stride 256).

Original:

```
REAL A(256,100), B(256,100)
...
DO I=1,256
  DO J=1,100
    A(I,J) = B(I,J) * 2
  END DO
END DO
```

Modified:

```
REAL A(257,100), B(257,100)   ! Column length 257
...
DO I=1,256
  DO J=1,100
    A(I,J) = B(I,J) * 2
  END DO
END DO
```

Memory accesses for the modified code are along the rows, but with a stride of one because each column length is now 257.

Amdahl's Law for vectorization

3.5

It is common for programs to be 70% to 80% vectorized; that is, spend 70% to 80% of their running time executing vector instructions. Although a vectorized loop runs 10 to 20 times faster than its scalar equivalent, a fully vectorized program typically executes 2 to 4 times faster than the original scalar code. The proportion of vectorization can be measured by hardware performance monitors, introduced on page 37.

The speedup of the whole program is lower than the speedup of a single loop because of a principle called Amdahl's Law, which states that the performance of a program is dominated by its slowest component. For a vectorized program, the slowest component is scalar code. A formulation of this law for vector code, which is R times faster than scalar code, is shown in the following equation:

$$s_v = \frac{1}{f_s + \frac{f_v}{R_v}}$$

In the equation,

s_v = maximum expected speedup from vectorization

f_v = fraction of a program that is vectorized

f_s = fraction of a program that is scalar = $1 - f_v$

R_v = ratio of scalar to vector processing time

For Cray Research systems, R_v ranges from 10 to 20; for the following discussion, assume a value of 10. For a program that is 50% vectorized, this formula gives a speedup of only 1.8. For a program that is 80% vectorized, the speedup is still only 3.6. Only with 100% vectorization can a program actually achieve a speedup of 10 (that is, the result s_v).

It is not always easy to reach 70% to 80% vectorization in a program, and vectorizing beyond this level becomes increasingly difficult, normally requiring major changes to the algorithm. Consequently, many users stop their vectorization efforts once the vectorized code is running 2 to 4 times faster than scalar code.

The preceding formula assumes that all vector code is R_v times faster than equivalent scalar code, and that the overhead for vector code is insignificant. In reality, not all vector code is R_v times faster than scalar code, and short vector lengths further reduce this factor. Also, vector code has a small startup cost, so that, at vector lengths of two or three elements, scalar code is usually faster. A realistic speedup for a program that is 80% vectorized might be 3.0 rather than 3.6.

Specifying vectorization

3.6

This subsection summarizes directives and command-line options used in vectorization. As shown in the manual *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071, the following `cf77` command lines all result in vectorized code:

```
cf77 pfile.f        # Default vectorization level

cf77 -Zv pfile.f    # Increases vectorization

cf77 -Zp pfile.f    # Increased vectorization with Autotasking
```

The following command line disables vectorization:

```
cf77 -Wf"-o novector" pfile.f
```

FPP options 3.6.1

To specify an FPP option, such as generating an FPP listing file, enclose that option in quotation marks following option `-Wd`, as in the following:

```
cf77 -Zv -Wd"-l listfile" pfile.f
a.out
```

FPP options and directives are shown in the appendix beginning on page 107.

`-o` *arguments* 3.6.2

Vectorization is disabled by any `cf77 -Wf"-o` *option*`"` where *option* is `off`, `noscalar`, or `novector`. Other `-Wf"-o"` options relating to vectorization are the following:

Option	Effect
`recurrence`, `norecurrence`	Enables or disables vectorization of reduction loops, discussed on page 80. The default is `recurrence`.
`vsearch`, `novsearch`	Enables or disables vectorization of search loops, discussed beginning on page 61. The default is `vsearch`.
`zeroinc`, `nozeroinc`	`cf77 -Wf"-o zeroinc"` causes the compiler to assume that some loop counters (see page 21) might be incremented by zero for each pass through the loop and produces conditional vector code for these loops. This degrades performance. The default is `nozeroinc`.

Compiler directives 3.6.3

Source code directives used by the compiling phase are in the form `CDIR$ DIRECTIVE`. These are ignored if vectorization is disabled by the command line. Compiler vectorization directives are as follows:

Directive	Effect
`VECTOR`, `NOVECTOR`	These directives turn vectorization on and off. Either directive applies until the end of the program unit unless the opposite directive appears.

Directive	Effect
NEXTSCALAR	Disables vectorization only for the next DO loop between the directive and the end of the program unit.
IVDEP[SAFEVL=*n*]	Indicates that any dependencies can be ignored if the vector length does not exceed *n*; if SAFEVL= does not appear, all dependencies are ignored. See page 50; dependencies are discussed on page 39. IVDEP is used when it is known that any apparent dependencies will not cause invalid results if a loop is vectorized. FPP inserts this directive when its analysis shows that a loop can be vectorized safely. Loops that have been vectorized with the use of IVDEP are indicated by the notation Vi in Loopmark listings.
VFUNCTION	Declares that a vector version of an external function exists, written in CAL; function names are listed on the directive separated by commas.
SHORTLOOP	Allows the compiler to generate faster code when a loop's trip count is known to be 64 or less. See page 79.
VSEARCH, NOVSEARCH	Enables and disables, respectively, vectorization of all search loops until the opposite directive is encountered or until the end of the program unit. The default is VSEARCH. Search loops are discussed on page 61. These directives override `cf77 -Wf"-o novsearch"` and `-Wf"-o vsearch"`, discussed in the previous subsection.
RECURRENCE, NORECURRENCE	Enables and disables, respectively, vectorization of all reduction loops until the opposite directive is encountered or until the end of the program unit. The default is RECURRENCE. Reduction loops are discussed on page 80. These directives override `cf77 -Wf"-o norecurrence"` and `-Wf"-o recurrence"`, discussed in the previous subsection.

Listings
3.6.4

By default, the compiler does not write an output listing. You receive a listing with `-Wf"-e"` list options or by including the directive `CDIR$ LIST` or `CDIR$ CODE` in your source code. In working with vectorization, you should include the `-Wf"-em"` option to produce listings using the Loopmark feature. Some other listing options are as follows: `c`, common block storage; `g`, generated code with CAL equivalent; `x`, cross-references.

The name of the compiler's default listing file is your source file name with the extension `.l`.

Examples:

```
cf77 -Zv -Wf"-e mcx" pgm.f   # To file pgm.l

cf77 -Zv -Wf"-e mcx -l pgm.4" pgm.f   # To file pgm.4
```

Loopmark listing
3.6.4.1

The compiler's Loopmark feature, specified on the command line by `cf77 -Wf"-em"`, generates source listings that indicate the kind of optimization that has been achieved on each loop. An example is shown in Figure 6.

In Loopmark listings, primary loop types are: `V`: vector; `S`: scalar; `W`: unwound (page 25). Modifiers added to these types are: `b`: bottom loaded; `c`: vectorized with a computed maximum safe vector length (page 49); `i`: unconditionally vectorized with `CDIR$ IVDEP`; `r`: unrolled (see *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071); `s`: short vector loop; `v`: short safe vector loop.

FPP listings
3.6.4.2

When you specify `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`, the CF77 dependency analysis phase, FPP, performs transformations on your source program and produces new, modified Fortran source (without changing your source file). You can examine the new source code in the following ways:

- Specify `cf77 -Wd"-l` *filename*`"` to cause FPP to produce a listing in file *filename*. This listing shows in detail the actions taken by FPP; it is distinct from the compiler's listings.
- When you specify `cf77 -Zv` or `-Zp` with `-Wf"-em"`, the resulting source listing written by the compiler shows the code produced by FPP, rather than your original source.
- Specify `cf77 -ZV` or `cf77 -ZP` to save FPP's output in file *file*`.m`.

```
                          L O O P M A R K   L E G E N D
                          ----------------------------
PRIMARY LOOP TYPE            LOOP MODIFIERS
-----------------            --------------
S - scalar loop              b - bottom loaded
V - vector loop              c - computed safe vector length
W - unwound loop             i - unconditionally vectorized with an IVDEP
                             k - kernel scheduled
                             r - unrolled
                             s - short vector loop
                             v - short safe vector length

 1      1.                                SUBROUTINE LMARK
 2      2.                                DIMENSION A(100),B(100),C(100)
 3      3.                                COMMON // J,K,N
 4      4. S--------------------<         DO L = 1,10
 5      5. S                                PRINT*, A(L)
 6      6. S-------------------->         ENDDO
 7      7. Vc-------------------<         DO I = K,N
 8      8. Vc                               A(I) = A(I) + A(I-K)
 9      9. Vc-------------------->        ENDDO
10     10.                           CDIR$ IVDEP
11     11. Vir------------------<        DO M = 1,100
12     12. Vir                             C(M) = A(M-J)
13     13. Vir                             A(M) = B(M)
14     14. Vir------------------>        ENDDO
15     15.                               END

                  V E C T O R I Z A T I O N   I N F O R M A T I O N
                  -------------------------------------------------
*** *** FF8021 [vector] < LMARK, Line = 4, File = loopm, Line = 4 > :
               Loop starting at line 4 was not vectorized.  It contains
               an input/output operation.
*** *** FF8005 [vector] < LMARK, Line = 7, File = loopm, Line = 7 > :
               Loop starting at line 7 was vectorized with a computed
               maximum safe vector length.
*** *** FF8006 [vector] < LMARK, Line = 11, File = loopm, Line = 11 > :
               Loop starting at line 11 was vectorized because an IVDEP
               compiler directive was specified.

                  O P T I M I Z A T I O N   I N F O R M A T I O N
                  -----------------------------------------------
*** *** FF8135 [opt_info] < LMARK, Line = 11, File = loopm, Line = 11>:
               Loop starting at line 11 was unrolled 2 times.
```

Figure 6. Loopmark listing

Assessing vectorization
3.7

This subsection describes methods of assessing the extent of vectorization in your program: the Loopmark feature (described on page 34) indicates the kind of code that was generated for each loop, and a variety of tools monitor your program during execution, allowing you to identify areas where additional effort would be useful. The monitoring tools are described fully in *UNICOS Performance Utilities Reference Manual*, publication SR–2040.

Flowtrace
3.7.1

Flowtrace gathers timing and calling information about procedure calls during program execution and writes the information to a file. Flowtrace incurs CPU overhead and requires recompilation and reloading. Example:

```
cf77 -F pgm.f
./a.out
flowview -Luch > flow.report
```

Following are ways to trace only selected routines, or sections of routines, to lower Flowtrace overhead:

- To trace specific subprograms, insert directives in your source code. Each `CDIR$ FLOW` enables Flowtrace for one program unit only, and must be contained in that program unit.
- Insert calls to the `FLOWMARK` subroutine at the beginning and end of a code block for which you need separate tracing and timing. The first call includes a name for the code block (entered as a null-terminated character string); the second invocation uses 0 as an argument. You do not need to invoke Flowtrace on the compiler's command line to use `FLOWMARK` calls. A `FLOWMARK` block should use at least 50 microseconds per call.

`FLOWMARK` example:

```
      ...
      CALL FLOWMARK ('PHANTOM'L)
      DO 10, I=1,10
      ...
   10 CONTINUE
      CALL FLOWMARK (0)
```

Hardware monitor: `hpm` ***and Perftrace***
3.7.2

On systems other than CRAY-2 systems, a hardware performance monitor measures the usage of groups of system components during program execution. The `hpm` command gives you readings for your whole program, and Perftrace works with the Flowtrace feature to give readings by routine and code block. There are four groups of monitors, which must be specified separately. Examples in this subsection show all four groups used.

Reports for both `hpm` and Perftrace are generated by the `perfview` command. This can be entered with no arguments for interactive use, or with arguments to generate a report. The examples here show the most commonly used arguments for generating a report. The `perfview` utility evaluates the raw hardware measurements and provides commentary on the balance of operations in your program.

`hpm` *command*
3.7.2.1

The `hpm` command is used similarly to `time`, in that your program's name is included on the same command line. Output is sent to `stderr`, which can be redirected using shell notation. The following examples show, for the Bourne and C shells, program `a.out` being executed four times to be monitored by each of the monitor groups, followed by the `perfview` command to generate a report in file `report.out`.

Bourne shell:

```
hpm -r -g0 ./a.out 2> perf.data
hpm -r -g1 ./a.out 2>> perf.data
hpm -r -g2 ./a.out 2>> perf.data
hpm -r -g3 ./a.out 2>> perf.data
perfview -LBuchM > report.out
```

C shell:

```
hpm -r -g0 ./a.out 2> perf.0
hpm -r -g1 ./a.out 2> perf.1
hpm -r -g2 ./a.out 2> perf.2
hpm -r -g3 ./a.out 2> perf.3
cat perf.[0-3] | perfview -LBuchM - > report.out
```

Perftrace
3.7.2.2

Perftrace gives the same kind of statistics as those given by the `hpm` command, but broken down by program unit. Perftrace incurs CPU and operating system overhead; you can decrease system overhead by using selective tracing, as shown previously for Flowtrace. Perftrace is invoked by specifying the Flowtrace option on the compilation command line and loading the `libperf.a` library instead of the Flowtrace library. Example:

```
cf77 -F -lperf pgm.f
./a.out
perfview -LBMuch > perf.report
```

Profiling
3.7.3

The Profiling feature indicates the relative amount of execution time used by individual parts of your program, represented as segments of program memory. It produces finer-grained results than those from Flowtrace. Profiling uses this sequence: compile with debug option, load with `libprof.a` library, execute, `prof`, `profview`. Example:

```
cf77 Wf"-ez" -l prof pgm.f
env PROF_WPB=1 ./a.out
prof -x a.out > pgm.prof
profview pgm.prof
```

Options can also be entered on the `profview` command so that you can save the output. Example:

```
profview -LmhDc pgm.prof > pgm.report
```

Dependencies [4]

Dependencies [4]

A primary inhibitor of vectorization is known as a *dependency*, which occurs when results of an operation could differ between scalar and vector processing. The CF77 compiling system inhibits generation of vector code in these cases and indicates the problem in the listing.

A *recurrence* is a data dependency between loop iterations. This occurs when an expression in one loop iteration requires a value that was defined in a previous iteration. Normally, recurrences cannot be vectorized because the reordering of operations would yield incorrect results. Consider the following program:

```
      PROGRAM VECSCAL
      INTEGER IA(4),IB(4),IC(4)
      DATA IA/11,12,13,14/
     &     IB/21,22,23,24/
     &     IC/31,32,33,34/
C                    [Print initial values]
      DO I=2,4
        IB(I)=IA(I-1)
        IA(I)=IC(I)
      ENDDO
C                    [Print final values]
      END
```

This code is compiled into scalar code because of the recurrence caused by the subscript `(I-1)`. The printed result follows:

Initial values:

```
IA = 11  12  13  14
IB = 21  22  23  24
IC = 31  32  33  34
```

Final values following loop (`IC` *is unchanged):*

```
IA = 11  32  33  34
IB = 21  11  32  33
```

If the loop is converted to vector code, the vector result must agree with this scalar result. Preceded by `CDIR$ IVDEP`, the loop is compiled into vector code, giving the following result, which differs from scalar and is therefore invalid:

```
IA = 11  32  33  34
IB = 21  11  12  13
```

The order of operations used in scalar and vector processing is as follows.

Scalar processing:

```
IB(2)=IA(1)
IA(2)=IC(2)
IB(3)=IA(2)  ! IB(3) gets new IA(2)
IA(3)=IC(3)
IB(4)=IA(3)  ! IB(4) gets new IA(3)
IA(4)=IC(4)
```

Vector processing:

```
IB(2)=IA(1)
IB(3)=IA(2)  ! IB(3) gets old IA(2)
IB(4)=IA(3)  ! IB(4) gets old IA(3)
IA(2)=IC(2)
IA(3)=IC(3)
IA(4)=IC(4)
```

Because vector processing of the Fortran DO loop in this example would generate incorrect results, the loop is not vectorized.

A simple test for dependency

4.1

This subsection describes a test to determine whether two appearances of an array in a loop can create a dependency conflict. This test is easy to apply, but it does not always give correct results. Some situations in which this test fails are discussed later in this section. Also, a more rigorous and accurate test is described later.

The following example illustrates this test.

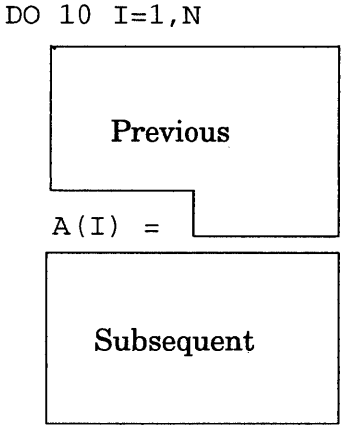
```
      DO 20 J = 2, M
         Z(J) = YY(J) + TEMPA
         R(J) = Z(J+1)/ TEMPB
20    CONTINUE
```

In the preceding loop, a potential for data dependency exists. The Z array is both defined and referenced within the loop.

Identify the two appearances as the *key definition* (where the Z array is defined) and the *other reference* (where the Z array is referenced). In the preceding DO loop, Z(J) is the key definition; Z(J+1) is the other reference.

Using the key definition and other reference, proceed with the following three steps:

1. Relative to the array's key definition, determine whether the other reference is in either the *previous area* or the *subsequent area*. The previous area includes the right side of the key definition statement, as follows:



   In this example, the other reference, `Z(J+1)`, is subsequent to the key definition, `Z(J)`.

2. Determine whether the subscript of the other reference is *greater* than or *less* than the subscript of the key definition.

   In this example, the subscript for `Z(J+1)` is greater than the subscript for `Z(J)`.

3. Determine whether the array subscripts are *incrementing* or *decrementing* on each iteration of the loop.

   In this example, the `DO` loop index `J` is incrementing on each iteration of the loop.

As shown in the preceding steps, the use of an array in a loop has the following characteristics:

- An array's other reference is either Previous or Subsequent to the array's key definition.
- The subscript on the array's other reference is either Greater or Less than on the array's key definition.
- The array's subscript is either Incrementing or Decrementing.

These three characteristics can be abbreviated, respectively, as P or S, G or L, and I or D. Using these abbreviations, there are a total of eight possibilities for loop-dependency analysis: PGI, PLI, PGD, PLD, SGI, SLI, SGD, and SLD. Four of these cases indicate dependencies that inhibit vectorization:

SLD SGI PLI PGD

The loop shown in the preceding example can then be described as subsequent, greater, and incrementing, or SGI, indicating that a data dependency may exist; vectorization is therefore inhibited.

The following example illustrates another case using the same test.

```
      DO 10 I = 1, 100
         X(I) = X(I+1)
10    CONTINUE
```

In the preceding code, `X(I)` is the key definition. `X(I+1)` is the other reference. The other reference is in the previous area. The subscript of the other reference is greater than the subscript of the key definition. The subscripts are incrementing on each iteration of the loop. This loop can be described as previous, greater, and incrementing, or PGI.

The following subsections contain examples of all eight types of loops, along with the vectorization message issued by the compiler for each loop.

Example codes that inhibit vectorization 4.1.1

The following shows an example of SGI (Subsequent, Greater, Incrementing) conflict:

```
1.       SUBROUTINE SGI(A, B, C)
2.       DIMENSION A(100), B(100), C(100)
3.       DO 10 I=1,99
4.          A(I) = B(I)
5.          C(I) = A(I+1)
6. 10    CONTINUE
7.       END

 cft77-8045 cf77: VECTOR SGI, Line= 3
        Loop at line 3 was not vectorized.
        It contains complex ordering of
        dependencies.
```

The following shows an example of SLD (Subsequent, Less, Decrementing) conflict:

```
1.       SUBROUTINE SLD(A, B, C)
2.       DIMENSION A(100), B(100), C(100)
3.       DO 10 I=100,2,-1
4.         A(I) = B(I)
5.         C(I) = A(I-1)
6. 10    CONTINUE
7.       END

cft77-8045 cf77: VECTOR SLD, Line= 3
        Loop at line 3 was not vectorized.
        It contains complex ordering of
        dependencies.
```

The following shows an example of PLI (Previous, Less, Incrementing) conflict:

```
 1.       SUBROUTINE PLI(A, B, C)
 2.       DIMENSION A(100), B(100), C(100)
 3.       DO 10 I=2,100
 4.       B(I) = A(I-1)
 5.       A(I) = C(I)
 6.    10 CONTINUE
 7.       END

cft77-8045 cf77: VECTOR PLI, Line=3
        Loop at line 3 was not vectorized.
        It contains complex ordering of
        dependencies.
```

The following shows an example of PGD (Previous, Greater, Decrementing) conflict:

```
1.        SUBROUTINE PGD(A, B, C)
2.        DIMENSION A(100), B(100), C(100)
3.        DO 10 I=99,1,-1
4.        B(I) = A(I+1)
5.        A(I) = C(I)
6.    10  CONTINUE
7.        END

cft77-8045 cf77: VECTOR PGD, Line=3
          Loop at line 3 was not vectorized.
          It contains complex ordering of
          dependencies.
```

Example codes that do not inhibit vectorization 4.1.2

The following example includes SGD (Subsequent, Greater, Decrementing) subscripts, which indicate no conflict:

```
1.        SUBROUTINE SGD(A, B, C)
2.        DIMENSION A(100), B(100), C(100)
3.        DO 10 I=99,1,-1
4.        A(I) = B(I)
5.        C(I) = A(I+1)
6.    10  CONTINUE
7.        END

cft77-8004 cf77: VECTOR SGD, Line=3
          Loop starting at line 3 was vectorized.
```

The following example includes SLI (Subsequent, Less, Incrementing) subscripts, which indicate no conflict:

```
1.        SUBROUTINE SLI(A, B, C)
2.        DIMENSION A(100), B(100), C(100)
3.        DO 10 I=2,100
4.        A(I) = B(I)
5.        C(I) = A(I-1)
6.    10  CONTINUE
7.        END

cft77-8004 cf77: VECTOR SLI, Line=3
          Loop starting at line 3 was vectorized.
```

The following example includes PLD (Previous, Less, Decrementing) subscripts, which indicate no conflict:

```
1.        SUBROUTINE PLD(A, B, C)
2.        DIMENSION A(100), B(100), C(100)
3.        DO 10 I=100,2,-1
4.        B(I) = A(I-1)
5.        A(I) = C(I)
6.    10  CONTINUE
7.        END

cft77-8004 cf77: VECTOR PLD, Line=3
         Loop starting at line 3 was vectorized.
```

The following example includes PGI (Previous, Greater, Incrementing) subscripts, which indicate no conflict:

```
1.        SUBROUTINE PGI(A, B, C)
2.        DIMENSION A(100), B(100), C(100)
3.        DO 10 I=1,99
4.        B(I) = A(I+1)
5.        A(I) = C(I)
6.    10  CONTINUE
7.        END

cft77-8004 cf77: VECTOR PGD, Line=3
         Loop starting at line 3 was vectorized.
```

A more rigorous test for dependency

4.2

This subsection shows a test for dependency analysis that is more rigorous than the test shown previously but which still does not cover all possible situations. It is the same test as that used by the compiling phase of CF77 to determine whether a loop has a data dependency. This test accounts for cases in which the dependency test described on page 41 is insufficient.

The following example is the same as that used to illustrate the simplified dependency analysis test, except that the loop control variable is incremented by 2, instead of 1, on each iteration.

```
      DO 20 J = 2, M, 2
        Z(J) = YY(J) + TEMPA
        R(J) = Z(J+1)/ TEMPB
   20 CONTINUE
```

In this code, there is a potential data dependency: the Z array is both defined and referenced in the loop.

1. As in the previous test, identify the two appearances as the *key definition* (where Z is defined) and the *other reference* (where Z is referenced). In the preceding loop, `Z(J)` is the key definition; `Z(J+1)` is the other reference.

2. Using the key definition and the other reference, let the "most previous" reference be *ref1*, and the "most subsequent" be *ref2*. It is critical to choose *ref1* and *ref2* accurately. The concepts of previous and subsequent are exactly the same as those defined in the simplified dependency analysis test.

3. Using *ref1* and *ref2*, define *index1*, *index2*, and *stride*, as follows:

   - *index1* = index of *ref1*. From the example, *index1* = index of `Z(J)`, which is `J`.
   - *index2* = index of *ref2*. From the example, *index2* = index of `Z(J+1)`, which is `J+1`.
   - *stride* = index *stride* (signed). From the example, *stride* =2.

4. If the sign of *index2* minus *index1* equals the sign of *stride*, there may be a dependency; proceed to step 5. Otherwise, no dependency exists.

   From the example:

   index1 = `J`, *index2* = `J+1`, *stride* = 2:

   - The sign of (*index2–index1*) = the sign of `((J+1)-(J))` = the sign of (1), which is positive.
   - The sign of (*stride*) = the sign of (2), which is positive.

   Because the sign of *index1* minus *index2* equals the sign of *stride* (both positive), there may be a dependency. Consequently, step 5 must be performed.

5. If (*index2* minus *index1*) mod *stride* equals 0, there is a dependency. Otherwise, no dependency exists.

   From the example:

   index1 = `J`, *index2* = `J+1`, *stride* = 2:

   - (*index1–index2*) mod *stride* = `((J+1) - (J))` mod 2 = 1 mod 2 = 1

   Because (*index2* minus *index1*) mod *stride* does not equal 0, there is no data dependency.

If the simplified dependency analysis test is used on the preceding code, the result is SGI. This implies that there is a possible data dependency. The more rigorous dependency test takes into account the stride of the indexes of the arrays.

Vectorizing recurrences

4.3

This subsection shows how a loop with a recurrence can be vectorized by preventing the recurrence from affecting the loop's result. Measures for resolving dependencies can be performed in either the compiling or dependency analysis phase.

When invoked with `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`, CF77's dependency analysis phase transforms the source code to produce new source that can be vectorized, without affecting your original source. Some capabilities described in the following subsections require the use of this phase, as noted.

Recurrence threshold

4.3.1

The *threshold* of a recurrence is the number of iterations that occur before a value is reused. If the vector length equals the threshold, the recurrence does not affect results in the vectorized version. A recurrence whose threshold exceeds 64 is fully vectorized. If the compiler can detect a threshold value of k in the range $2 < k < 64$, the loop is vectorized with a vector length of k.

```
1.          SUBROUTINE SHORT_VL(A)
2.          DIMENSION A(100)
3.          DO 20 I = 7,100
4.             A(I) = A(I-6) + 1.0
5.    20    CONTINUE
6.          END

cft77-8072 cf77: VECTOR SHORT_VL, Line=3
          Loop starting at line 3 was vectorized
          with a vector length of 6.
```

The recurrence in this code has a threshold of 6. Therefore, it is vectorizable with the shortened vector length of 6.

Safe vector length
4.3.2

To offer better performance on some loops that involve potential recurrences, the compiler can include a run-time test to determine a *safe vector length*. This length is less than or equal to the recurrence threshold, just discussed, assuring that the code's result will be unaffected by a recurrence. The threshold's value, *t*, need not be known at compile time; the safe vector length equals k if k < 64, and 64 otherwise. Note that the use of a safe vector length allows lengths of 1 and 2, which can degrade performance even to the point of being slower than scalar code.

If you compile using `cf77 -Wf"-em"` to obtain Loopmark listings, a `c` notation to the left of the code indicates the use of a computed safe vector length. In the following example, the vector length is a function result unknown at compile time:

```
1.            PROGRAM SAFEVL
2.            DIMENSION A(-100:100),B(100),C(100)
3.            K = KFUN(A(I))
4.            N = NFUN(B(I))
5. Vc----<    DO I = K,N    ! N is vector length
6. Vc           A(I) = A(I) + A(I-K)
7. Vc---->    ENDDO
8.            END

cft77-8005 cf77: VECTOR SAFEVL, Line=5
Loop starting at line 5 was vectorized with a
computed maximum safe vector length.
```

In the next example, the vector length is the result of an expression:

```
2.            SUBROUTINE RUNTIME(A,B,C)
3.            DIMENSION A(-100:100),B(100),C(100)
4.            COMMON // J
5. Vc----<    DO I = 1,100
6. Vc           C(I) = A(I-J)
7. Vc           A(I) = B(I)
8. Vc---->    ENDDO
9.            END

cft77-8005 cf77: VECTOR RUNTIME, Line=3
Loop starting at line 5 was vectorized with a
computed maximum safe vector length.
```

In the preceding Fortran code:

- If ($J \geq 64$), the loop fully vectorizes with a vector length of 64.
- If ($J < 1$), the loop fully vectorizes with a vector length of 64.
- If ($1 \leq J < 64$), the loop vectorizes with a vector length of J.

In the preceding code, variable J is the ambiguous parameter that causes the use of a run-time safe vector length.

Options for safe vector length
4.3.2.1

Some loops are too complex to allow a run-time test for vector length. If you know that the vector length in such a loop is safe, you can force vectorization by use of CDIR$ IVDEP, listed on page 33.

Even when a safe vector length is long enough to make vectorization worthwhile (that is, 3 or above), the use of a run-time test involves significant overhead. Therefore, you should consider modifying each loop that performs poorly due to the use of a run-time test for vector length. You can do one of the following:

- Suppress vectorization with CDIR$ NEXTSCALAR (applying to only one loop) or CDIR$ NOVECTOR.
- Insert CDIR$ IVDEP [SAFEVL=*n*] before the loop, where *n* is a vector length that you know to be safe. If you omit SAFEVL=*n*, you must assure that the recurrence will not invalidate your program's result.

Conditional vectorization
4.3.3

Although the use of a run-time test for safe vector length, described in the preceding subsection, has sometimes been called *conditional vectorization*, in CF77 this term refers to another technique: the generation of both scalar and vector code, one of which is selected and branched to at run time.

A run-time test can be either coded into your source or added by CF77 at the source level, when invoked with cf77 -Zv, -ZV, -Zp, or -ZP. The test added to the source controls a branch to one of two copies of the loop. CDIR$ NEXTSCALAR preceding one of the copies suppresses vectorization (of a DO loop), causing the code to be translated into scalar operations. CDIR$ IVDEP preceding the other copy causes it to ignore dependencies.

When the `cf77 -Zv` option is used, CF77 can generate two versions of a loop, together with a run-time test. If the loop is not data dependent, a vectorized version of the loop executes; otherwise a scalar version executes.

Example:

```
      SUBROUTINE POTNTL ( A, B, IP1, N )
      REAL A(*), B(*)
      DO 10 I = 1,N
         A(IP1+I) = A(I) + B(I)  ! Potentially dependent
 10   CONTINUE
```

Translation (compiled with `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`):

```
      IF ( IP1.LE.0 .OR. IP1.GE.N ) THEN
CDIR@ IVDEP
        DO 10 I = 1, N    ! "Vector" loop
           A(IP1+I) = A(I) + B(I)
 10     CONTINUE
      ELSE
CDIR@ NEXTSCALAR
        DO 77001 I = 1, N    ! "Scalar" loop
           A(IP1+I) = A(I) + B(I)
77001   CONTINUE
      ENDIF
```

You can disable this transformation by using `cf77 -Wd"-dm"` or the `NOALTCODE` directive.

Reference reordering 4.3.4

Sometimes a loop does not contain true feedback, but still cannot be vectorized as is because of the order in which array elements are referenced. In cases like the following example, CF77 creates a temporary variable to hold the "old" elements of an array so that they are still available when needed later. This allows the loop to be safely vectorized.

Example:

```
      SUBROUTINE TEST (A,B,C,D,E,N)
      DIMENSION A(N), B(N), C(N), D(N), E(N)
      DO 10 I = 1, N
         A(I) = B(I) + C(I) + D(I)
         D(I) = E(I) + A(I+1)
  10  CONTINUE
      RETURN
      END
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      REAL R1S
      DIMENSION A(N), B(N), C(N), D(N), E(N)
CDIR@ IVDEP
      DO 10 I = 1, N
         R1S = A(1+I)
         A(I) = B(I) + C(I) + D(I)
         D(I) = E(I) + R1S
   10 CONTINUE
      RETURN
      END
```

CF77 also reorders entire statements to eliminate dependencies.

Ambiguous subscript resolution
4.3.5

When the information contained in a loop is insufficient to determine the storage relationship between two array references, the situation is called *ambiguous subscripting* or *potential feedback*. In such situations, when invoked with `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`, CF77 recognizes that a loop does not feed back data between iterations. This is performed in the dependency analysis phase: FPP searches for statements outside the loop that may resolve the ambiguity and inserts an `IVDEP` directive before each inner loop it knows to be free of data dependence; this ensures that these loops are vectorized.

In the following loop, CF77 finds the assignment to `L`, which makes it clear that `LW` can never equal `L`, and it recognizes that there is no feedback.

Example:

```
      L = LW - 1
      DO 2 I = 1, N          ! Not vectorized
         Y(L) = Y(L) + X(I)*Y(LW)
         LW = LW + 1
  2   CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      L = LW - 1
CDIR@ IVDEP
      DO 2 I = 1, N          ! Unconditionally vectorized
         Y(L) = Y(L) + X(I)*Y(LW)
         LW = LW + 1
  2   CONTINUE
```

Loop splitting and peeling
4.3.6

When invoked with `cf77 -Zv, -ZV, -Zp, or -ZP`, CF77 can in some cases determine that a dependency is limited to a subset of the operations in a loop, and will modify the loop to remove or isolate the dependency. The `%DP` field of the loop summary in an FPP listing (enabled as shown on page 34) indicates how much of the loop is left unvectorized. If more than a certain percentage of a loop is dependent, the loop is not vectorized. (The exact percentage depends on other factors.)

Splitting to remove recurrence
4.3.6.1

When possible, CF77 splits data-dependent loops into vectorizable and nonvectorizable parts. You can disable loop splitting by using the `cf77 -Wd"-ds"` option.

Example:

```
      DO 2530 L = 2, NLAYM1    ! Does not vectorize
      SDPOL(L,M) = SDPOL(L-1,M) + CONVPL(L) -
     1      DSIG(L)*PITPOL(M)
      IF (COMG) OMEGA(1,JKP,L) = SDPOL(L,M)*1.E6
 2530 CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
CDIR@ IVDEP
      DO 2530 L = 1, J2S     ! Vectorizes
         R1V(L) = CONVPL(1+J1S+L) -
     1        DSIG(1+J1S+L)*PITPOL(M)
 2530 CONTINUE
      DO 77002 L = 1, J2S      ! Does not vectorize
         SDPOL(1+J1S+L,M) = SDPOL(J1S+L,M)+R1V(L)
77002 CONTINUE
CDIR@ IVDEP
      DO 77003 L = 1, J2S    ! Vectorizes
         IF (COMG) OMEGA(1,JKP,1+J1S+L) =
     1        SDPOL(1+J1S+L,M)*1.E6
77003 CONTINUE
```

Splitting to avoid a recurrent point
4.3.6.2

CF77 recognizes certain special cases, involving the need to process the loop in two pieces to avoid an interior point that may cause dependency problems.

Example:

```
      DO 100 I = 1, 100
         A(I) = A(101-I) + B(I)
 100  CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
CDIR@ IVDEP
      DO 10 I = 1, 50
         A(I) = A(101-I) + B(I)
   10 CONTINUE
CDIR@ IVDEP
      DO 77001 I = 1, 50
         A(50+I) = A(51-I) + B(50+I)
77001 CONTINUE
```

Loop peeling
4.3.6.3

When CF77 recognizes that data dependency is caused by one or a few iterations at one end of the `DO` index range, it can use *loop peeling* to remove the dependency. This rewrites one or more iterations as single statements outside of the loop.

Example:

```
      DO 100 K = 1, 30
         A(K) = A(1) + B(K)
 100  CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      A(1) = A(1) + B(1)
CDIR@ IVDEP
      DO 100 K = 1, 29
         A(1+K) = A(1) + B(1+K)
 100  CONTINUE
```

If the first iteration is peeled from the loop (executed as a single scalar statement), the rest of the loop iterations can be vectorized.

Up to three iterations can be peeled from the beginning or end of a loop. The loop may be of variable length, but the stride (increment) must be a constant.

Last value saving
4.3.7

When CF77 restructures a loop, it may eliminate the definition of some vector indexes (integer variables that are incremented by a constant amount each pass through a loop). When necessary, the final values of such variables are recreated. CF77 examines the flow of the program unit to see whether the final value of an index variable is used outside the loop; if not (as is usually the case), then the vectorized code does not store a value into the index variable.

In at least one case, CF77 may unnecessarily save last values. This occurs when index variables are in common storage and there are calls to external subprograms. When this happens, CF77 cannot be sure that the variables are not used through common storage by other program units, and it must generate values for them.

Example:

```
      COMMON /BLOCK/ J
      ...
      DO 23 I = 1,N
        J = J + 1
        ...
   23 CONTINUE
      CALL SUBX          ! Does SUBX use J?
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      ...
   23 CONTINUE
      IF (N .GT. 0) THEN      ! Must be > 0 iterations
        J = J + N             ! Save last value
      ENDIF
      CALL SUBX
```

If such variables are not used by other program units, use `cf77 -Wd"-du"`, or the `NOLSTVAL` directive to suppress the generation of last values.

Using data dependency directives

4.4

This subsection describes data dependency directives that you can use to provide CF77 with additional information so that code can be fully optimized. Some data dependency directives also have command line equivalents; see Table 6, page 116. Data dependency directives are also discussed beginning on page 123.

Directives in this subsection are interpreted by FPP, which is activated by `cf77` options `-Zv`, `-ZV`, `-Zp`, or `-ZP`. The directives take the form `CFPP$` *directive scope*, where `L` indicates loop scope, `R` indicates routine scope, and `F` indicates file scope. See page 115.

Declaring nonrecurrence: NODEPCHK

4.4.1

The `CFPP$ NODEPCHK` directive lets you direct CF77 to ignore potential data dependencies in a loop. This capability should be used only when you know that no real recurrence exists. When it detects potential feedback, CF77 issues a message asking you to apply this directive if the loop is not recurrent.

The meaning of NODEPCHK is similar to that of IVDEP; they differ in that NODEPCHK is used in the dependency analysis phase, whereas IVDEP is used in the compiling phase. In addition, IVDEP is written by FPP itself to communicate with the compiler.

The following is an example where you might want to disable data dependency checking. In this loop, CF77 cannot be sure that N1 does not equal N2 and thus rejects the loop:

Example:

```
      SUBROUTINE MOVE (A, B, N, N1, N2)
      REAL A(N,*), B(*)
      DO 3 I = 1,N                ! Not vectorized
  3      A(I+1,N1) = A(I,N2)+B(I)
```

If you know that N1 is never equal to N2, you can insert a directive as shown in the following:

```
CFPP$ NODEPCHK
      DO 4 I = 1,N                ! Vectorized
  4      A(I+1,N1) = A(I,N2)+B(I)
```

Suppress equivalence checking: NOEQVCHK 4.4.2

Recurrences can be hidden by the use of EQUIVALENCE statements. Although this problem is rare, CF77 examines EQUIVALENCE statements to find hidden recurrences, and it suppresses any potentially unsafe transformations. Equivalencing of many variables can inhibit vectorization of most loops in a program.

Example:

```
      SUBROUTINE TEST (A,B,N)
      DIMENSION A(N), B(N)
      DO 10 I = 2, N
         TEMP = A(I-1) + A(I-2)
         B(I) = TEMP + 1.0/TEMP
         A(I) = SQRT(B(I)) - 5.0
  10  CONTINUE
      RETURN
      END
```

In the preceding loop, the reference to A at the top of the loop conflicts with the store into A at the bottom. FPP prints a message to this effect and inserts a directive to explicitly inhibit vectorization. For this loop, FPP gives the following diagnostic:

```
fpp-37 cf77: COMMENT TEST, Line = 4, File = fppmsg.f, Line = 4
 Feedback of elements of array 'A'. Optimization inhibited.
 Conflict on line 6. The DO index is 'I', the DO label is '10'.
```

The NOEQVCHK directive indicates to CF77 that EQUIVALENCE statements can be ignored for data dependency analysis; that is, variables with different names do not overlap in storage.

In the following example, several local arrays have been equivalenced to a large array in common (perhaps to save space). If the arrays could overlap (for instance, if the value of the variable N was 1500 in the DO 100 loop), the DO 100 loop cannot be vectorized. But, if we know the arrays do not overlap, the NOEQVCHK directive can be applied to the whole routine.

Example:

```
      COMMON /BIG/ POOL(100000)
      DIMENSION A(1),B(1),C(1)
      EQUIVALENCE (POOL(1),A(1)),(POOL(1001),
     1      B(1)),(POOL(2001),C(1))
CFPP$ NOEQVCHK R   ! Ignore equivalences
      ...
      DO 100 I = 1, N
            A(I+IA) = B(I+IB) + C(I+IC)
 100  CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
CDIR@ IVDEP
      DO 100 I = 1, N          ! Vectorized
         A(I+IA) = B(I+IB) + C(I+IC)
 100  CONTINUE
```

Relationship between variables: **RELATION** 4.4.3

You can use the CFPP$ RELATION directive to provide additional information to CF77 about array subscript ranges; this information helps determine whether a loop is safe to vectorize. The RELATION directive has the following form:

```
CFPP$ RELATION ( simple1 .rel. simple2 )
```

simple1 and *simple2* are simple integer variables (one of them can be an integer constant), and *rel* is one of GT, LT, GE, LE, EQ, or NE, with the normal Fortran meanings.

When CF77 cannot otherwise determine whether the relationship between two uses of an array is recurrent, it searches the RELATIONs supplied by the user for the current routine to see whether they help.

RELATION directives are informative only; they do not force any action. They can be applied at the loop, routine, or file level. If conflicting relations are given, the result is unpredictable. You must ensure that the relations specified are correct and consistent.

Example:

```
CFPP$ RELATION ( J.GE.N )  R
      ...
      DO 100 I = 1, N    ! If J .GE. N, no overlap
        A(I+J) = A(I) + B(I)
 100  CONTINUE
```

The RELATION directive is provided for situations in which you are unsure whether the NODEPCHK directive (a blanket assertion of nonrecurrence) is safe, or know it is not, but have some information about relative values of index variables.

Safe indirect addressing: PERMUTATION 4.4.4

When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, feedback is possible even if the subscript is identical. Feedback occurs if there are any repeated elements in the subscripting array.

Sometimes, indirect addressing is used because the elements of interest in an array are sparsely distributed; in this case, an integer array is used to point at the elements that are really desired, and there are no repeated elements in the integer array (as in the following example).

This information can be passed to CF77 through the CFPP$ PERMUTATION directive, which asserts that the specified integer arrays contain no repeated elements (that is, they serve merely to permute the elements of the arrays they indirectly address).

The format of the PERMUTATION directive is as follows:

```
CFPP$ PERMUTATION ( ia1, ia2, ... , ian )
```

PERMUTATION declares that the integer arrays (*ia1*, etc.) do not have repeated values for the entire routine.

Example:

```
CFPP$ PERMUTATION (IPNT)   ! IPNT has no repeated values.
      ...
      DO 100 I = 1, N
         A(IPNT(I)) = A(IPNT(I)) + B(I)
 100  CONTINUE
```

Loops Containing `IF` Statements [5]

Loops Containing IF Statements [5]

The CF77 compiling system tries to vectorize loops containing conditional statements and branches. IF statements can be used to form the loop itself (IF...GOTO) or can be contained within a loop. Loops with conditional statements are subject to the same rules as other loops, as well as some additional rules.

IF loops and search loops

5.1

IF loops and search loops have the same characteristics for vectorizing.

An IF loop uses the IF...GOTO construct to control the repeated execution of a series of statements. Example:

```
      REAL A(N)
      I=1
    3 CONTINUE
      A(I) = I**2
      I = I + 1
      IF (I.LT.N) GOTO 3
```

A *search loop* is a DO loop that can be exited by means of an IF...GOTO construct. Example:

```
      DO 10, I=1,N
      ...
      IF (A(I).LE.B(I)) GOTO 20
      ...
   10 CONTINUE
   20 ...
```

Search loops and IF loops, including those with complex exit conditions, are normally vectorized.

To be vectorized, a search loop must satisfy the following requirements, relating to scalar-vector equivalence:

- The loop must be executed the correct number of times.
- The correct exit must be taken.
- All scalar values must be correct when an exit is taken.

The following subsections discuss aspects of these requirements.

Loops that meet these requirements are vectorized as follows: the IF expression is evaluated for the full set of loop iterations indicated by the loop's DO statement; the point where the IF expression is satisfied indicates the vector length to be used for the other expressions within the loop. This length is used when those expressions are executed.

The VSEARCH and NOVSEARCH directives control vectorization of search loops as a class, when vectorizing such loops would give invalid results. The two primary cases when CDIR$ NOVSEARCH might be needed are: the IF test expression would create an illegal value if applied to array addresses past the search value; and the value being tested for the loop exit could be the array index. These directives are described more fully in *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071.

Early exits 5.1.1

An *early exit* is any exit from the body of a loop; that is, any exit that decreases the number of loop iterations from the trip count calculated for the loop. A loop can have more than one early exit, provided that no requirements described in this section are violated.

Example:

```
      DO 10, I=1,N
      IF (A(I).GE.B(I+1)) GOTO 30
      ...
      IF (A(I).LT.B(I)) GOTO 20
      ...
   10 CONTINUE
   20 ...
      ...
   30 ...
```

Note that early exits, particularly multiple early exits, contribute to the complexity of a loop, discussed on page 27.

Compound exit conditions 5.1.2

The compound logical operators, .OR. and .AND., do not inhibit vectorization of search loops. These are also called *short-circuit* operators.

The following code shows an example of a search loop with compound logical operators:

```
1.        SUBROUTINE 2_EXITS(A,B,TEST1,TEST2,N)
2.        LOGICAL TEST1(N), TEST2(N)
3.        DO 10 I=1,N
4.          IF(TEST1(I) .OR. TEST2(I)) GO TO 99
5.  10    CONTINUE
6.  99    PRINT *, I
7.        END

cft77-8004 cf77: VECTOR 2_EXITS, Line=3
          Loop starting at line 3 was vectorized.
```

Branches and trip counts 5.1.3

Because an early exit affects the number of loop iterations, it also affects the number of array elements to be processed; this number must be the same with vector code as it would be with scalar code. In the following examples, a value is computed that limits how many values of array C can be modified; this follows the assignment to C(I).

1. All early exits from a loop must appear in the first block of the loop; that is, early exits must appear before the first branch, such as an IF statement. (Blocks are discussed on page 20.)

   Because vector processing is the processing of elements in groups, the group of elements to be processed by a vectorized statement must be determined before the statement can be executed. In the following example, each conditional statement causes uncertainty about the group of elements to be processed by the other conditional statement. Therefore, the loop is not vectorized.

```
   DO 7 I = 1,N                  ! Vectorization inhibited
     IF (A(I).NE.0) C(I) = 1     ! Branch
     IF (B(I).NE.0) GOTO 8       ! Exit follows branch
 7 CONTINUE
 8 ...
```

2. The assignment in the following example is unconditional, but two exits follow it. Determining which exit will take effect (that is, on which iteration the exit will occur, determining how many elements are processed) cannot currently be done in vector code for this case.

```
      DO 3 I = 1,N          ! Vectorization inhibited
        C(I) = 1
        IF (A(I).NE.0) GOTO 4
        IF (B(I).NE.0) GOTO 4
    3 CONTINUE
    4 ...
```

Moving early exits: code motion
5.1.4

The compiler can move a loop's final conditional exit (within the intermediate text) to the top of the block containing the exit so that the exit precedes the block's vectorizable statements. This compiler technique, called *code motion*, allows the loop to be vectorized because the number of iterations is known before the vectorizable statements are executed. (The number of iterations is determined by the branch statement at run time.)

As discussed on page 20, a *block* is a section of code with no explicit or implicit branches, where a branch indicates code that may or may not be executed, including simple cases such as `IF(L) I = 1.`

You can predict the compiler's use of code motion by considering the fundamental requirement for any vectorization: the result must be equivalent to that given by scalar code. Only one exit can be moved, and the move must not move the exit before a branch (that is, to a different block), so that the number of iterations is known before vectorized code is executed. Other requirements are as follows:

- A value that is needed for the test to be moved is computed before the test.
- A value that is computed before the exit is carried out of the loop.
- There is a store of a value ahead of the exit.
- An indirect memory reference (gather) is needed to compute the exit condition.

Examples in which the compiler uses code motion:

1.

```
      DO 5 I = 1,N
        C(I) = 1
        IF (B(I).NE.0) GOTO 6 ! Exit to be moved
    5 CONTINUE
    6 ...
```

2.

```
      DO 2 I = 1,N
        IF (A(I).NE.0) GOTO 3 ! Branch defines block
        C(I) = 1
        IF (B(I).NE.0) GOTO 3 ! Exit to be moved
    2 CONTINUE
    3 ...
```

Code motion does not apply to examples 1 and 2 beginning on page 63. In example 1, the GOTO would move from after the conditional assignment to before it, thereby crossing a block boundary (which is not allowed with code motion); in example 2, two exits would need to be moved, but this is not currently done.

Values computed following exit 5.1.5

The condition for an early exit must not depend on values computed (in a previous iteration) in the portion of the loop following the exit. Example:

```
      DO 8 I = 1,N
        IF (A(I).GT.100) GOTO 9
        A(I+1) = A(I+1) + B(I)   ! Inhibits vectorization
    8 CONTINUE
    9 ...
```

Although more complicated loops may present a challenge, you can reorder this example relatively easily, as follows:

```
      IF (A(1).GT.100) THEN
        I = 1
      ELSE
        DO 6 I = 2,N              ! Vectorization permitted
          A(I) = A(I) + B(I-1)
          IF (A(I).GT.100) GOTO 7
    6   CONTINUE
    7   IF (I.EQ.N+1) A(N+1) = A(N+1) + B(N)
      ENDIF
```

Search loop with two exits
5.1.6

The following examples show loops containing two exits; one loop is vectorized and the other is not. The code containing a store, `B(I)=A(I)`, can be vectorized only if the number of iterations is known before the vector operation.

Vectorized:

```
 1.            SUBROUTINE SRCH(A,B,N,IFND,T1,T2)
 2.            REAL A(N), B(N)
 3.            LOGICAL T1(N), T2(N)
 4. V---<      DO 10 I=1,N
 5. V            IF(T1(I)) GO TO 20
 6. V            IFND = I
 7. V            IF(T2(I)) GO TO 20
 8. V            B(I) = A(I)   ! Vector tore after exits
 9. V--->10    CONTINUE
10.       20  CONTINUE
11.           END

cft77-8004 cf77: VECTOR SRCH, Line=4
         Loop at line 4 was vectorized.
```

Not vectorized:

```
 1.      SUBROUTINE E2_NOVEC(A,B,N,IFOUND,T1,T2)
 2.      REAL A(N), B(N)
 3.      LOGICAL T1(N), T2(N)
 4.      DO 10 I=1,N
 5.        B(I) = A(I)    ! Vector store before exits
 6.        IF(T1(I)) GO TO 20
 7.          IFOUND = I
 8.        IF(T2(I)) GO TO 20
 9. 10   CONTINUE
10. 20   CONTINUE
11.      END

cft77-8017 cf77: VECTOR E2_NOVEC, Line=4
        Loop at line 4 was not vectorized.
        A value must be stored before an
        exit condition is computed.
```

The compiler message shown here indicates that code to evaluate an exit condition must be generated at the top of the loop. Any code preventing the compiler from moving the exit test ahead of all stores prevents vectorization.

Indirect addressing 5.1.7

If an exit condition involves indirect addressing, vectorization is inhibited.

```
  DO 7 I = 1,N
    IF ((A(K(I))).NE.0.0) GOTO 8 ! Inhibits vectorization
7 CONTINUE
8 ...
```

An operand range error could occur during execution of the vectorized version of the loop if values in `K` exceeding the index when the `GOTO` is executed are not appropriate indices for `A`.

Branches
5.2

This subsection describes conditions that determine whether loops containing branches can be vectorized.

Branches into a loop
5.2.1

If any statement branches into the loop from outside it, the loop cannot be vectorized. (Such a branch into any DO loop is prohibited; therefore, this rule applies only to IF loops.) Example:

```
      I = 0
      IF (C.NE.0) GOTO 2     ! Inhibits vectorization
      I = 1
    1 CONTINUE
        A(I) = 0
    2   B(I) = 0
        I = I + 1
      IF (I.LT.N) GOTO 1
```

Any loop whose body contains a branch destination is not mentioned in messages concerning vectorization. You can usually eliminate such a branch, at the cost of duplicating some code.

Loops with backward branches
5.2.2

If a loop contains a backward branch, the loop cannot be vectorized. A backward branch is itself a loop, and any backward branch that is useful needs to be explicitly structured as a loop.

Example:

```
 5. S------<       DO 7, I = 1,N
 6. S                A(I) = B(I,10)
 7. S V----<     6   J = J + 1    ! Innermost loop
 8. S V              B(I,J) = (REAL(I))/(REAL(J))
 9. S V---->         IF (J.LT.10) GOTO 6
10. S------>     7 CONTINUE
```

In other cases, the backward branch may be a poor way of coding for the desired result or may be incorrectly structured to be vectorized. Example:

```
 6. S------<       DO 7 I = 1,10
 7. S                A(I) = 0
 8. S S----<6        B(I) = B(I) - 1
 9. S S---->         IF (B(I).GT.10) GOTO 6
10. S------>7      CONTINUE

 cft77-8035 cf77: VECTOR BACKBRANCH, Line = 6
   Loop starting at line 6 was not vectorized.
   It contains an inner loop.
 cft77-8014 cf77: VECTOR BACKBRANCH, Line = 8
   Loop starting at line 8 was not vectorized.
   The loop control test uses floating-point,
   double precision or complex arithmetic.
```

The inner loop shown here uses a real value, `B(I)`, as its index and cannot be vectorized. Even if this loop is modified so as to vectorize, the outer loop, `DO 7`, will fail to vectorize.

Loops with forward branches 5.2.3

Forward branches inside loops do not prevent vectorization.

```
      DO 3 I = 1,N
        IF (A(I).GE.0) GOTO 2   ! Permits vectorization
        A(I) = 0
    2   B(I) = 0
    3 CONTINUE
```

This branch would normally be coded with the `IF` directly controlling the assignment to `A(I)`.

Obsolete IF *statements* 5.2.4

Because the obsolete three-branch arithmetic `IF` statement, by definition, has multiple destinations, it prevents vectorization of loops. The following code includes a loop containing an arithmetic `IF`:

```
 1.        SUBROUTINE ARITH_IF(A,B,C,K1)
 2.        REAL A(100), B(100), C(100)
 3.        INTEGER K1(100)
 4.        DO 100 I = 1, 100
 5.          IF (K1(I)) 50,40,30
 6.  30         A(I) = 0.
 7.             GOTO 60
 8.  40         B(I) = 0.
 9.             GOTO 60
10.  50         C(I) = 0.
11.  60       CONTINUE
12.  100    CONTINUE
13.        END

cft77-8025 cf77: VECTOR ARITH_IF, Line=4
        Loop starting at line 4 was not
        vectorized.  It contains an
        arithmetic IF statement.
```

Conditional block examples

5.3

A piece of the body of a loop that may be executed on a particular iteration of the loop is called a *conditional block*. The following examples show loops containing a variety of code constructs within conditional blocks.

The following code shows several conditional blocks:

```
 1.        SUBROUTINE COND_BLOCKS (A,B,C,K1)
 2.        REAL A(100), B(100), C(100)
 3.        INTEGER K1(100)
 4.        DO 100 I = 1, 100
 5.           IF (K1(I).GT.0) THEN
 6.              A(I) = 0.
 7.           ELSEIF (K1(I).EQ.0) THEN
 8.              B(I) = 0.
 9.           ELSE
10.              C(I) = 0.
11.           ENDIF
12.  100    CONTINUE
13.        END

cft77-8004 cf77: VECTOR COND_BLOCKS, Line=4
        Loop starting at line 4 was vectorized.
```

The following code shows nested conditional blocks:

```
 1.       SUBROUTINE NEST (A,B,C,D,T1,T2,T3)
 2.       REAL A(100), B(100), C(100), D(100)
 3.       LOGICAL T1(100),T2(100),T3(100)
 4.       DO 100 I = 1, 100
 5.          IF (T1(I)) THEN
 6.             IF (T2(I)) THEN
 7.                A(I) = 0.
 8.             ELSE
 9.                B(I) = 0.
10.             ENDIF
11.          ELSE
12.             IF (T3(I)) THEN
13.                C(I) = 0.
14.             ELSE
15                 D(I) = 0.
16.             ENDIF
17.          ENDIF
18. 100   CONTINUE
19.       END

cft77-8004 cf77: VECTOR NEST, Line=4
        Loop starting at line 4 was vectorized.
```

The following example shows a forward branch:

```
    1.        SUBROUTINE FORWARD_BRANCH(TEST,A)
    2.        REAL A(100)
    3.        LOGICAL TEST(100)
    4.        DO 100 I = 1, 100
    5.          IF (TEST(I)) GOTO 50
    6.            A(I) = 0.
    7. 50       CONTINUE
    8. 100    CONTINUE
    9.        END

cft77-8004 cf77: VECTOR FORWARD_BRANCH, Line=4
        Loop starting at line 4 was vectorized.
```

The CF77 compiling system can also vectorize the following code constructs:

- Reductions in a conditional block.
- Loop counters incremented or decremented in a conditional block.

The following example shows a loop counter conditionally incremented:

```
 1.        SUBROUTINE LIV_INC(A,B,TEST,INDX)
 2.        REAL A(100), B(100)
 3.        INTEGER INDX
 4.        LOGICAL TEST(100)
 5.        INDX = 1
 6.        DO 100 I = 1, 100
 7.           IF (TEST(I)) THEN
 8.              A(INDX) = B(I)
 9.              INDX = INDX + 1
10.           ENDIF
11.  100  CONTINUE
12.       END

cft77-8004 cf77: VECTOR LIV_INC, Line=6
        Loop starting at line 6 was vectorized.
```

The following example shows a reduction in a conditional block:

```
1.      SUBROUTINE COND_RED(A,SUMPOS)
2.      REAL A(100)
3.      SUMPOS = 0.
4.      DO 10 I = 1,100
5.        IF (A(I).GT.0.0) SUMPOS = SUMPOS + A(I)
6.  10  CONTINUE
7.      END

cft77-8004 cf77: VECTOR COND_RED, Line=4
        Loop starting at line 4 was vectorized.
```

Source transformations of IF statements

5.4

When invoked with the `cf77 -Zv` option, the CF77 compiling system's first phase, FPP, performs transformations on your source code before the compile step. This does not affect your original program. This analysis and rearrangement of code can remove constructs that inhibit vectorization. This subsection describes the kinds of transformations that are performed.

Converting IF *loops to* DO *loops*

5.4.1

Innermost IF loops are converted into DO loops under certain conditions. An innermost IF loop is one that contains no other IF or DO loops. To be convertible, the loop must meet the following criteria:

1. The loop must have a single entrance and a single exit.
2. The iteration count for the loop must be determinable at execution time before the loop is entered.

Example:

```
      SUBROUTINE IFLOOP (N, JB, A, B, S)
      REAL A(N), B(N)
C
      J = JB
   10 CONTINUE
      A(J) = 0.0
      J = J + 1
      IF ( J .GT. N ) GO TO 20
      B(J) = S
      GO TO 10
   20 CONTINUE
```

Translation:

```
      ...
      INTEGER J1X
C
      J = JB
   10 CONTINUE
      A(J) = 0.0
      J = J + 1
      J1X = J
      IF (N - J1X + 1 .GT. 0) THEN
CDIR@    IVDEP
         DO 77001 J = 1, N - J1X + 1
            B(J1X+J-1) = S
            A(J1X+J-1) = 0.0
77001    CONTINUE
      ENDIF
   20 CONTINUE
```

The `cf77 -Wd"-d l"` option disables conversion of `IF` loops to `DO` loops. All directives (such as `NODEPCHK`) affecting `IF` loops must have routine or global scope.

Tests on the loop index are converted into restricted-range loops; see page 75.

Arithmetic `IF`s are converted to block `IF`s. Example:

```
      DO 500 I = 1, N
         IF (A(I)) 400,420,400
400      B(I) = A(I)
         GO TO 500
420      C(I) = A(I)
500   CONTINUE
```

Translation:

```
CDIR@ IVDEP
      DO 500 I = 1, N
         IF (A(I) .NE. 0) THEN
             B(I) = A(I)
         ELSE
             C(I) = A(I)
         ENDIF
500   CONTINUE
```

Conditional operations 5.4.2

CF77 can analyze any combination of conditional assignments, conditional and unconditional forward branching (including arithmetic IFs and computed GOTOs with four or less labels), and block IFs. Because of compilation-speed restrictions, there is a limit of six simultaneously active conditions.

The loop below shows a conditional assignment that depends on the loop index. CF77 eliminates such conditions by adjusting the limits of the vector operation.

Example:

```
      DO 1013 I = 1,100
         IF (I.NE.J) A(I) = B(I) + C(I)
1013  CONTINUE
```

Translation:

```
      IF (J-1.GE.0 .AND. J-1.LE.99) THEN
CDIR@ IVDEP
         DO 1013 I = 1, J-1
            A(I) = B(I) + C(I)
 1013    CONTINUE
CDIR@ IVDEP
         DO 77001 I = J+1, 100
            A(I) = B(I) + C(I)
77001    CONTINUE
      ELSE
CDIR@ IVDEP
         DO 77002 I = 1, 100
            A(I) = B(I) + C(I)
77002    CONTINUE
      ENDIF
```

The test, IF (J-1.GE.0...), determines whether J is in the range of the DO loop index. If so, the operation is performed up to the J-1 element and then from the J+1 element to N. If J is not in the range of the DO loop index, the operation is performed on all elements.

A condition dependent on the loop index is of the general form:

```
IF ( index expression.rel.invariant expression )
```

In the previous IF statement, variables are as follows:

Variable	Description
index expression	This must take the following form: *vector index * (invariant expression) + or - (invariant expression)*
rel	A relational operator; possible values are EQ, NE, GT, GE, LT, or LE.
invariant expression	Represents an arithmetic expression, constant, or variable.

Example:

```
          DO 100 I = 1, N
             A(I) = A(I) + 1.
             IF (I.EQ.K) B(I) = A(I)/C(I)
             D(I) = B(I) + A(I)
     100  CONTINUE
```

Translation:

```
CDIR@ IVDEP
      DO 100 I = 1, N
         A(I) = A(I) + 1.
 100  CONTINUE
      IF (K.GE.1 .AND. K.LE.N) B(K) = A(K)/C(K)
CDIR@ IVDEP
      DO 77001 I = 1, N
         D(I) = B(I) + A(I)
77001 CONTINUE
```

Conditional reductions 5.4.3

CF77 uses the CVMGT function to convert certain conditional reductions into a vectorizable form, if the vectorizable expression that is being reduced contains only +, -, *, count, and logical operators.

Example:

```
          DO 400 I = 1, N
             IF ( A(I) .GT. EPSILN ) S = S + A(I)
     400  CONTINUE
```

Translation:

```
CDIR@ IVDEP
      DO 400 I = 1, N
         S = S + CVMGT(A(I),0.0,A(I).GT.EPSILN)
400   CONTINUE
```

Conditional reductions are not transformed if they contain operators with singularities (for example, square root). Singularities cause problems because CVMGT evaluates the expression for every element of the vector. The IF statement protects the evaluation of the expression, as shown in the following example:

```
      DO 400 I=1,N
         IF (B(I) .NE. 0.0)S = S + A(I)/B(I)
400   CONTINUE
```

Loops [6]

Loops [6]

This section discusses issues concerning special classes of loops, modification of loop nests, and special loop optimizations.

Special cases
6.1

Some iterative code segments do not fit the criteria for full vectorization presented in preceding sections of this manual, yet they can be vectorized. For these code segments, the generated machine code consists of both vector and scalar instructions. This section presents cases in this category.

Short vector loops
6.1.1

A short vector loop is a fully vectorizable loop with an iteration count less than or equal to 64, thus permitting the entire loop to be executed with vector instructions and no looping construct.

```
1.          SUBROUTINE FEWTRIPS(A,T)
2.          DIMENSION A(20)
3.          DO 10 I = 1,20
4.            A(I) = A(I) + T
5.    10    CONTINUE
6.          END

cft77-8003 cf77: VECTOR FEWTRIPS, Line=3
           Loop starting at line 3 was a short
           vector loop.
```

The SHORTLOOP directive, listed on page 33, can be placed before a loop if the compiler will not be able to determine the number of iterations, but you can assure that the number will be 64 or less.

Reduction loops 6.1.2

A *reduction loop* reduces an array to a scalar value by doing a cumulative operation on all of the array's elements; this involves including the result of the previous iteration in the expression of the current iteration. A reduction can be vectorized if the operator is one of the arithmetic operators +, –, *, /; MAX or MIN; or one of the logical operators.

Example:

```
      SUM = 0.0
      DO 4 I = 1,N
        SUM = SUM + A(I)     ! Permits vectorization
    4 CONTINUE
```

A reduction is a special case of vectorization; the generated code effectively computes partial sums by using vector instructions, then collapses the partial sums with vector and scalar instructions to produce the final scalar result. The compiler can vectorize DO loops containing one of the following types of reductions, where S is a scalar variable and *e* is any expression that does not inhibit vectorization:

```
S = S + e              S = S*e
S = S - e              S = MIN(S,e)
S = S .AND. e          S = S .OR. e
S = MAX(S,e)           S = S/e
```

The compiler can also vectorize DO loops containing split reductions such as the following:

```
S = S + e1
S = S + e2
```

If a reduction loop has a low trip count that cannot be determined at compile time, the cost of vectorizing it might exceed the benefit. The NORECURRENCE directive allows you to specify scalar execution for loops of this class (within a range of code or a whole subprogram), without the need for multiple NOVECTOR directives. You can prevent vectorization of all reduction loops in an entire compilation by compiling with `cf77 -Wf"-o norecurrence"`.

Implied DO *loops*
6.1.3

An implied DO loop can be used as the argument list in a READ or WRITE statement.

Example:

```
1.        SUBROUTINE IMPLIED_IO(A)
2.        REAL A(100)
3.        READ(10) (A(I),I=1,100)
4.        END
cft77-1004 cf77: VECTOR IMPLIED_IO
     Loop starting at line 3 was vectorized.
```

The resultant code makes a call to a special entry point in the I/O library that vectorizes portions of the I/O operation. Although the I/O statement does not vectorize, the implied DO loop within the statement does.

Array reference with constant index
6.1.4

An array reference with a constant index should be equivalent to a scalar reference, but currently it is not handled as simply as a true scalar because it can confuse dependence analysis.

1. This loop contains no recurrence because I never takes the value 1. The compiler establishes this by comparing the bounds on I with 1.

```
   DO 10 I = 2,N,2
10 A(I) = A(1)          ! Permits vectorization
```

2. In this loop, the absence of a recurrence is not revealed by a comparison of the bounds on I with 2, so the compiler does not detect the absence. If this loop is preceded by CDIR$ IVDEP, it vectorizes.

```
   DO 10 I = 1,N,2
10 A(I) = A(2)          ! Inhibits vectorization
```

Loop nest restructuring
6.2

When invoked with cf77 -Zv, -ZV, -Zp, or -ZP, CF77 examines all IF and DO loops within a nest of loops, for possible optimization. If an outer loop is more suitable than the innermost for vectorization, CF77 exchanges the two loops. The amount of data dependence, size of array strides, and vector length are the criteria used to choose the "best" loop. Data

dependency messages include the label and the index of the loop to which they refer, which helps determine the exact location of the dependency.

Nest analysis 6.2.1

Unless you direct otherwise, CF77 analyzes a loop nest in these steps:

1. Loops are examined from innermost loops outward, until CF77 finds a loop or loops appropriate for vectorization.
2. If a nontranslatable construct is encountered (for example, a READ statement), analysis of any loops further out in the nest is disabled.

You can modify this procedure by using directives (for example, SELECT, NOCONCUR, or NOVECTOR) and/or setting switches. See Table 6, page 116, for a list of CF77 user directives.

Selection criteria 6.2.2

To select the best loop in a nest for vectorization, CF77 uses the following criteria:

- Loop iteration count
- Stride size of array references
- Percentage of data dependent code
- Percentage of conditionally executed code

In general, longer vectors, smaller strides, less dependence, and less conditionality are favored.

Example:

```
      DO 10 I = 1, M
      DO 10 J = 1, M
10       A(I,J) = B(I,J)**2
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      DO 77001 J = 1, M
CDIR@ IVDEP
        DO 77002 I = 1, M
           A(I,J) = B(I,J)**2
77002   CONTINUE
77001 CONTINUE
```

Example:

```
       COMMON /BLOCK/ PRESSURE(3,47),
      1  TEMPERATURE(3,47), VOLUME(3,47)
       DO 10 J = 1, NLAT
          DO 20 K = 1, NDEPTH
             PRESSURE(K,J) = CONST*VOLUME(K,J)*
     1                TEMPERATURE(K,J)
   20    CONTINUE
   10 CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      DO 10 K = 1, NDEPTH
CDIR@ IVDEP
        DO 77001 J = 1, NLAT
           PRESSURE(K,J) = CONST*VOLUME(K,J)*
     1               TEMPERATURE(K,J)
77001   CONTINUE
   10 CONTINUE
```

You can use the `cf77 -Wd"-et"` option to increase the relative weight of the factors listed in the preceding text, particularly the weight of small stride relative to the CF77 bias toward picking the original innermost loop for vectorization. Invoking this switch can result in loops being interchanged more often.

Loop optimizations 6.3

CF77 performs loop optimizations as described in the following subsections.

Loop collapse 6.3.1

Loop nests that traverse all of the inner dimensions of the arrays in the loop can be automatically "collapsed" into single loops with larger iteration counts.

Collapse criteria are as follows:

- The loops must be tightly nested, with one loop index per array dimension.
- The inner loop bounds must be identical to the array bounds (see K in the following example).
- All the vector array references in the loops must conform; that is, have the same subscripting.

Example:

```
      SUBROUTINE DWEEB ( L,M,N,A,B )
      DIMENSION A(L,M,N), B(L,M,N)
      DO 100 K = 2, N-1
         DO 100 J = 1, M
            DO 100 I = 1, L
               A(I,J,K) = B(I,J,K)
100         CONTINUE
```

Translation (compiled with `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`):

```
CDIR@ IVDEP
      DO 77001 K = 1, L*M*(N-2)
         A(K,1,2) = B(K,1,2)
77001 CONTINUE
```

Loop fusion 6.3.2

CF77 combines consecutive loops that have no statements between them and that give the same answers when merged. This aids other loop optimizations and reduces loop overhead.

Example:

```
      DO 311 I = 1,100
         A(I) = B(I) + SQRT(C(I))
  311 CONTINUE
      DO 312 I = 1,100
         E(I) = B(I) + (C(I))**2
  312 CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      DO 311 I = 1,100
         A(I) = B(I) + SQRT(C(I))
         E(I) = B(I) + (C(I))**2
  312 CONTINUE
```

Source-level loop unrolling 6.3.3

Loop unrolling makes a copy of a loop's body for each loop iteration and replaces the original loop with these copies, to be executed as straight-line code. Loop unrolling benefits nonvector loops whose scalar optimization is inhibited because they have too few operations per pass. Loop unrolling reduces the percentage of time spent in loop overhead, and provides more instructions for the optimizer to overlap in each pass of the loop; it can also aid other loop optimizations.

Note

The compiler uses a technique similar to that described here, but within the intermediate text. When performed by the compiler, this feature is called *loop unwinding*. Another compiler feature is called *loop unrolling*, but this is distinct from the FPP feature of the same name; see *CF77 Compiling System, Volume 1: Fortran Reference Manual*, publication SR–3071. The compiler performs this optimization by default, whereas FPP must be enabled to do it. Enabling this technique at the source level can be advantageous for loops with low iteration counts.

FPP has two modes of loop unrolling: automatic and explicit. The `CFPP$ UNROLL` directive controls both of these modes (see page 122). When used with routine or file scope (`R` or `F`), `UNROLL/NOUNROLL` enables or disables automatic unrolling. When used with local scope (`L` or blank), `UNROLL` directs CF77 to explicitly unroll the following loop. Automatic unrolling can also be enabled using `cf77 -Wd"-ef"`.

When automatic loop unrolling is enabled, FPP unrolls inner loops that satisfy these criteria:

- The vector length is constant, and below the vector threshold.

- The vector length times the number of statements in the loop is less than 32.
- The loop contains only assignment statements. No branches, I/O statements, or external references are allowed.
- The DO loop control parameters are integer.
- The last value of the loop index is not required after the loop is executed.

These restrictions do not apply to loops unrolled explicitly. The only inhibitors in this case are assigned GOTOs and I/O keywords other than END=, ERR=, FMT=, and UNIT=. In explicit mode, outer loops can also be unrolled.

The following is an example of automatic unrolling of a loop with a small fixed iteration count; the loop is completely unrolled into three assignment statements.

Example:

```
CFPP$ UNROLL R
      ...
      DO 311 I = 1, 3
         D(I) = A(I) + B(I)*C(I)
  311 CONTINUE
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
D(1) = A(1) + B(1)*C(1)
D(2) = A(2) + B(2)*C(2)
D(3) = A(3) + B(3)*C(3)
```

For more information about the UNROLL directive, see "Transformation directives," page 118.

Translation of array notation 6.3.4

CF77 translates array section syntax, a CF77 extension, into DO loops, which can then be vectorized and/or autotasked.

Example:

```
A(IB:IE,JB:JE) = B(IB:IE,JB:JE) + C(IB:IE,JB:JE)
D(IB:IE,JB:JE) = SQRT(B(IX(IB:IE),JB:JE))
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
      DO 77001 J2X = 1, JE - JB + 1
CDIR@ IVDEP
         DO 77002 J1X = 1, IE - IB + 1
            A(J1X-1+IB,J2X-1+JB) = B(J1X-1+IB,
     1         J2X-1+JB) + C(J1X-1+IB,J2X-1+JB)
            D(J1X-1+IB,J2X-1+JB) = SQRT(B(IX
     2          (J1X-1+IB),J2X-1+JB))
77002    CONTINUE
77001 CONTINUE
```

Conversion of array syntax can be disabled by using the `cf77 -Wd"-d1"` (not `1`) option.

`VFUNCTION` *directive use* 6.3.5

CF77 recognizes the CFT77 `VFUNCTION` directive and treats such functions as vector intrinsics. A function that has been specified on a `VFUNCTION` directive does not inhibit optimization by CF77.

Example:

```
CDIR$ VFUNCTION JOE
      DO 777 I = JB, JE
 777     A(I) = JOE(B(I))
```

Translation (compiled with `cf77 -Zv, -ZV, -Zp, or -ZP`):

```
CDIR$ VFUNCTION JOE
CDIR@ IVDEP
      DO 777 I = JB, JE
 777     A(I) = JOE(B(I))
```

Vectorization Examples [7]

Vectorization Examples [7]

This section contains loopmark listings and vectorization messages for a variety of Fortran code segments. Loopmark listings are specified on the command line in the form `cf77 -Wf"-em"`.

Each loop in a Loopmark listing is identified with a single uppercase letter (the *primary loop type* and, in some cases, a single lowercase letter (the *loop modifier*). Loops are identified according to the following legend:

Primary loop type:

- `S` Scalar loop
- `V` Vector loop
- `W` Unwound loop

Loop modifiers:

- `b` Bottom loaded
- `c` Computed safe vector length
- `i` Unconditionally vectorized with `CDIR$ IVDEP`
- `k` Kernel scheduled
- `r` Unrolled
- `s` Short vector loop
- `v` Short safe vector length

Each Fortran code segment shown was translated by the Fortran preprocessor FPP, specified by `cf77 -Zv`. Where FPP translated the code sequence to enhance vectorization, the FPP translations and resulting vectorization information are shown. For the other code segments, FPP did not find any additional possibilities for vectorization.

```
  1.                      PROGRAM SEQ1
  2.                      DIMENSION A(100), B(100), C(100)
  3.                      ADD(X,Y) = X + Y
  4. V-------------<      DO 10 I = 100, 1, -1
  5. V                       A(I) = ADD(B(I), C(I))
  6. V------------->   10 CONTINUE
  7.                      PRINT *, A
  8.                      END

cft77-8004 cf77: VECTOR SEQ1, Line=4
              Loop starting at line 4 was vectorized.

  1.                      PROGRAM SEQ2
  2.                      DIMENSION A(100), B(100), C(100)
  3.                      DATA B, C/100*3.0, 100*8.0/
  4. S-------------<      DO 10 I = 1, 100
  5. S                      A(I) = ADD(B(I), C(I))
  6. S------------->   10 CONTINUE
  7.                      PRINT *, A
  8.                      END

cft77-8020 cf77: VECTOR SEQ2, Line=4
              Loop starting at line 4 was not vectorized.
              It contains a subroutine call.

  1.                      FUNCTION ADD(X,Y)
  2.                      ADD = X + Y
  3.                      END

  1.                      PROGRAM SEQ3
  2.                      DIMENSION A(100)
  3.                      DATA XX / 50 /
  4. Vs------------<      DO 10 I = 1, 50
  5. Vs                     A(I) = TAN(XX)
  6. Vs------------->   10 CONTINUE
  7.                      PRINT *, A
  8.                      END

cft77-8003 cf77: VECTOR SEQ3, Line=4
              Loop starting at line 4 is a short vector loop.

  1.                      PROGRAM SEQ4
  2.                      DIMENSION A (100), B(100)
  3. Vs------------<      DO 10 I = 10, 50
  4. Vs                   IF (A(I).GT.B(I)) A(I) = B(I)
  5. Vs------------->   10 CONTINUE
  6.                      PRINT *, A
  7.                      END

cft77-8003 cf77: VECTOR SEQ4, Line=3
              Loop starting at line 3 is a short vector loop.
```

```
 1.                      PROGRAM SEQ5
 2.                      DIMENSION A(64),B(64),C(64),D(64)
 3.                      PARAMETER (NA=100)
 4.                      CALL BOB(A,B,C,D,NA)
 5.                      PRINT *, A
 6.                      CALL JILL(A, B, NA)
 7.                      PRINT *, A
 8.                      END

 1.                      SUBROUTINE BOB(A,B,C,D,NA)
 2.                      REAL A(NA), B(NA), C(NA), D(NA)
 3. V------------<       DO 30 I = 1, NA
 4. V                      IF(C(I).GE.0) THEN
 5. V                       IF (D(I).EQ.0) THEN
 6. V                         A(I) = B(I) / A(I)
 7. V                       ELSE
 8. V                         A(I) = B(I) / A(I)
 9. V                       ENDIF
10. V                      ENDIF
11. V------------>    30 CONTINUE
12.                     RETURN
13.                     END

cft77-8004 cf77: VECTOR BOB, Line=3
               Loop starting at line 3 was vectorized.

 1.                      SUBROUTINE JILL(A,B,NA)
 2.                      REAL A(NA)
 3.                      REAL B(NA)
 4. V------------<       DO 30 I = 1, NA
 5. V                      IF(A(I).GE.0) THEN
 6. V                       IF (A(I).EQ.0) THEN
 7. V                         A(I) = B(I)
 8. V                       ELSE
 9. V                         A(I) = B(I) / A(I)
10. V                       ENDIF
11. V                      ENDIF
12. V------------>    30 CONTINUE
13.                     RETURN
14.                     END

cft77-8004 cf77: VECTOR JILL, Line=4
               Loop starting at line 4 was vectorized.
```

```
  1.                      PROGRAM SEQ6
  2.                      DIMENSION A(100)
  3.                      PARAMETER (KK = 1)
  4. S------------<       DO 10 J = 1, 10
  5. S Vs---------<         DO 10 I = 20, 30
  6. S Vs                     A(J+KK * (I-1)) = 0.0
  7. S-Vs--------->    10 CONTINUE
  8.                      PRINT *, A
  9.                      END

cft77-8035 cf77: VECTOR SEQ6, Line=4
               Loop starting at line 4 was not vectorized.
               It contains an inner loop.
cft77-8003 cf77: VECTOR SEQ6, Line=5
               Loop starting at line 5 is a short vector loop.

  1.                      PROGRAM SEQ7
  2.                      DIMENSION A(100,100,100)
  3. S------------<       DO 10 I = 5, 95
  4. S S----------<        DO 10 J = 10, 90
  5. S S V--------<          DO 10 K = 1, 100
  6. S S V                     A(I, J, K) = A(I+4, J-4, K)
  7. S-S-V-------->    10 CONTINUE
  8.                      PRINT *, A
  9.                      END

cft77-8035 cf77: VECTOR SEQ7, Line=3
               Loop starting at line 3 was not vectorized.
               It contains an inner loop.
cft77-8035 cf77: VECTOR SEQ7, Line=4
               Loop starting at line 4 was not vectorized.
               It contains an inner loop.
cft77-8004 cf77: VECTOR SEQ7, Line=5
               Loop starting at line 5 was vectorized.

  1.                      PROGRAM SEQ8
  2.                      DIMENSION A(100)
  3. Vs-----------<       DO 10 I = 10, 60
  4. Vs                     A(I-3) = A(I+1)
  5. Vs----------->    10 CONTINUE
  6.                      PRINT *, A
  7.                      END

cft77-8003 cf77: VECTOR SEQ8, Line=3
               Loop starting at line 3 is a short vector loop.
```

```
  1.                      PROGRAM SEQ9
  2.                      DIMENSION A(100), B(100,100), C(100)
  3. Vs-----------<       DO 10 I = 1, 50
  4. Vs                    B(I) = A(I)
  5. Vs                    C(I) = B(I,7)
  6. Vs----------->    10 CONTINUE
  7.                      PRINT *, C
  8.                      END

cft77-8003 cf77: VECTOR SEQ9, Line=3
               Loop starting at line 3 is a short vector loop.

  1.                      PROGRAM SEQ10
  2.                      DIMENSION A(100), B(100), K(10)
  3.                      DATA B/100*1/, K/1,2,3,4,5,6,7,8,9,10/
  4. V------------<       DO 10 N = 1, 100
  5. V                      B(N) = 1.
  6. V------------>    10 CONTINUE
  7. Sb-----------<       DO 20 LIV = 2, 11
  8. Sb                     K(LIV) = K(LIV-1)
  9. Sb----------->    20 CONTINUE
 10. Vs-----------<       DO 30 J = 1, 10
 11. Vs                     I = J + K(2)
 12. Vs                     A(I) = B(J)
 13. Vs----------->    30 CONTINUE
 14.                      PRINT *, A
 15.                      END

cft77-8004 cf77: VECTOR SEQ10, Line=4
               Loop starting at line 4 was vectorized.
cft77-8044 cf77: VECTOR SEQ10, Line=7
               Loop starting at line 7 was not vectorized.
               It contains a recurrence on K at line 8.
cft77-8003 cf77: VECTOR SEQ10, Line=10
               Loop starting at line 10 is a short vector loop.

  1.                      PROGRAM SEQ11
  2.                      DIMENSION A(100), B(100), C(100)
  3. V------------<       DO 10 I = 90, 1, -1
  4. V                      A(I) = B(I+2)
  5. V                      A(I-3) = C(I)
  6. V------------>    10 CONTINUE
  7.                      PRINT *, A
  8.                      END

cft77-8004 cf77: VECTOR SEQ3, Line=3
               Loop starting at line 3 was vectorized.
```

```
  1.                     PROGRAM SEQ12
  2.                     DIMENSION A(100,100)
  3. S-----------<       DO 10 J = 5, 95
  4. S Sbr-------<       DO 10 I = 90, 10, -1
  5. S Sbr                 A(I,J) = A(I+1,J)
  6. S-Sbr-------->   10 CONTINUE
  7.                     PRINT *, A
  8.                     END

cft77-8035 cf77: VECTOR SEQ12, Line=3
               Loop starting at line 3 was not vectorized.
               It contains an inner loop.
cft77-8044 cf77: VECTOR SEQ12, Line=4
               Loop starting at line 4 was not vectorized.
               It contains a recurrence on A at line 5.
```

FPP translated the previous Fortran code segment as follows:

```
  1.                     PROGRAM SEQ12
  2.               C...Translated by FPP  5.0 (3.03M1) 07/15/91  12:57:28
  3.                     DIMENSION A(100,100)
  4. S-----------<       DO I = 1, 81
  5. S             CDIR@    IVDEP
  6. S V----------<         DO J = 1, 91
  7. S V                      A(91-I,4+J) = A(92-I,4+J)
  8. S V---------->         END DO
  9. S----------->       END DO
 10.                     PRINT *, A
 11.                     END

cft77-8035 cf77: VECTOR SEQ12, Line=4
               Loop starting at line 4 was not vectorized.
               It contains an inner loop.
cft77-8006 cf77: VECTOR SEQ12, Line=6
               Loop starting at line 6 was vectorized. An
               IVDEP compiler directive was specified.
```

```
  1.                      PROGRAM SEQ13
  2.                      INTEGER A(200)
  3.                      J = 184
  4. S-------------<      DO 10 I = 185, 200
  5. S                      A(I) = A(J)
  6. S                      J = J - 1
  7. S------------->   10 CONTINUE
  8.                      PRINT *, A
  9.                      END

cft77-8044 cf77: VECTOR SEQ13, Line=4
              Loop starting at line 4 was not vectorized.
              It contains a recurrence on A at line 5.
```

FPP translated the previous Fortran code segment as follows:

```
  1.                      PROGRAM SEQ13
  2.                C...Translated by FPP 5.0 (3.03M1) 07/15/91  13:03:22
  3.                      INTEGER A(200)
  4.                      J = 184
  5.                CDIR@ IVDEP
  6. Vs------------<      DO 10 I = 185, 200
  7. Vs                     A(I) = A(J)
  8. Vs                     J = J - 1
  9. Vs------------->   10 CONTINUE
 10.                      PRINT *, A
 11.                      END

cft77-8003 cf77: VECTOR SEQ13, Line=6
              Loop starting at line 6 is a short vector loop.
```

```
  1.                     PROGRAM SEQ14
  2.                     DIMENSION A(100,2)
  3.                     PARAMETER (N=100)
  4.                     NA=100
  5.                     K = -1
  6.                     CALL BOB(A,NA)
  7.                     PRINT *, A
  8.                     END

  1.                     SUBROUTINE BOB(A,NA)
  2.                     REAL A(NA,2)
  3. V-------------<     DO 10 I = 1, 100
  4. V                     A(I+K,1) = A(I+K,1) + 1.0
  5. V                     A(I+K,2) = A(I+K,2) + 1.0
  6. V------------->  10 CONTINUE
  7.                     RETURN
  8.                     END

cft77-8004 cf77: VECTOR BOB, Line=3
               Loop starting at line 3 was vectorized.

  1.                     PROGRAM SEQ15
  2.                     DIMENSION A(300), B(300), C(300)
  3. Sr------------<     DO 10 J = 1, 298
  4. Sr                    A(J) = B(J)
  5. Sr                    C(J) = A(J+1)
  6. Sr------------>   10 CONTINUE
  7.                     PRINT *, C
  8.                     END

cft77-8045 cf77: VECTOR SEQ15, Line=3
               Loop starting at line 3 was not vectorized.
               It contains complex ordering of dependencies.
```

FPP translated the previous Fortran code segment as follows:

```
  1.                     PROGRAM SEQ15
  2.                C...Translated by FPP 5.0 (3.03M1) 07/15/91  13:07:52
  3.                     DIMENSION A(300), B(300), C(300)
  4.                CDIR@ IVDEP
  5. Vr------------<     DO 10 J = 1, 298
  6. Vr                     C(J) = A(1+J)
  7. Vr                     A(J) = B(J)
  8. Vr------------>   10 CONTINUE
  9.                     PRINT *, C
 10.                     END

cft77-8004 cf77: VECTOR SEQ15, Line=5
               Loop starting at line 5 was vectorized.
```

```
  1.                      PROGRAM SEQ16
  2.                      DIMENSION A(300), B(300), C(300)
  3.                      EQUIVALENCE (JJ, KK)
  4. Sr-----------<       DO 10 JJ = 300, 5, -1
  5. Sr                     A(KK) = C(KK)
  6. Sr                     B(KK) = A(KK-4)
  7. Sr----------->    10 CONTINUE
  8.                      PRINT *, B
  9.                      END

cft77-8051 cf77: VECTOR SEQ16, Line=4
               Loop starting at line 4 was not vectorized.
               It contains a scalar store on JJ.

  1.                      PROGRAM SEQ17
  2.                      DIMENSION A(1000)
  3. Sr-----------<       DO 10 I = 1,999
  4. Sr                     A(I+1) = A(I)
  5. Sr----------->    10 CONTINUE
  6.                      PRINT *, A
  7.                      END

cft77-8044 cf77: VECTOR SEQ17, Line=3
               Loop starting at line 3 was not vectorized.
               It contains a recurrence on A at line 4.

  1.                      PROGRAM SEQ18
  2.                      DIMENSION A(100), B(100)
  3. V------------<       DO 10 I = 1, 100
  4. V                      A(I) = 4.0
  5. V                      J = I + 4
  6. V                      B(I) = A(J)
  7. V----------->    10 CONTINUE
  8.                      PRINT *, B
  9.                      END

cft77-8004 cf77: VECTOR SEQ18, Line=3
               Loop starting at line 3 was vectorized.

  1.                      PROGRAM SEQ19
  2.                      DIMENSION B(50)
  3. Vs-----------<       DO 10 I = 1, 20, 2
  4. Vs                     B(I) = B(I-1)
  5. Vs----------->    10 CONTINUE
  6.                      END

cft77-8003 cf77: VECTOR SEQ19, Line=3
               Loop starting at line 3 is a short vector loop.
```

```
  1.                      PROGRAM SEQ20
  2.                      DIMENSION IA(5000)
  3.                      INTEGER IR
  4.                      CALL SUB1(IA, IR)
  5.                      PRINT *, IR
  6.                      END

  1.                      SUBROUTINE SUB1(IA, IR)
  2.                      DIMENSION IA(5000)
  3. Vs-----------<       DO 80 I = 1, 64
  4. Vs                     IR = IR - 66 - IA(I)
  5. Vs----------->    80 CONTINUE
  6.                      RETURN
  7.                      END

cft77-8003 cf77: VECTOR SEQ20, Line=3
              Loop starting at line 3 is a short vector loop.

  1.                      PROGRAM SEQ21
  2.                      PARAMETER (N = 100)
  3.                      DIMENSION X(N), Y(N), Z(N), VX(N), II(N)
  4. S------------<       DO 1 I = 1, N
  5. S                      X(I) = X(II(I))
  6. S----------->     1  CONTINUE
  7.                      PRINT *, X
  8.                C
  9. VVr---------->       Y(:) = Y(II(:))
 10.                      PRINT *, Y
 11.                C
 12. Sr----------<        DO 2 I = 2, N-1
 13. Sr                     Z(I) = Z(I-1) + Z(I + 1)
 14. Sr----------->    2  CONTINUE
 15.                      PRINT *, Z
 16.                C
 17. VVr---------->       VX(2:N-1) = VX(1:N-2) + VX(3:N)
 18.                      PRINT *, VX
 19.                      END

cft77-8061 cf77: VECTOR SEQ21, Line=4
              Loop starting at line 4 was not vectorized.
              The subscripts are ambiguous on X.
cft77-8004 cf77: VECTOR SEQ21, Line=9
              Loop starting at line 9 was vectorized.
cft77-8004 cf77: VECTOR SEQ21, Line=9
              Loop starting at line 9 was vectorized.
cft77-8044 cf77: VECTOR SEQ21, Line=12
              Loop starting at line 12 was not vectorized.
              It contains a recurrence on Z at line 13.
cft77-8004 cf77: VECTOR SEQ21, Line=17
              Loop starting at line 17 was vectorized.
cft77-8004 cf77: VECTOR SEQ21, Line=17
              Loop starting at line 17 was vectorized.
```

FPP translated the previous Fortran code segment as follows:

```
 1.                 PROGRAM SEQ21
 2.                                 PARAMETER (N = 100)
 3.          C...Translated by FPP 5.0 (3.03M1) 07/15/91  13:14:07
 4.                 DIMENSION X(N), Y(N), Z(N), VX(N), II(N)
 5.                 INTEGER J1X
 6.                 REAL R1X(100),R2X(98)
 7. S-------<       DO 1 I = 1, N
 8. S                 X(I) = X(II(I))
 9. S------->    1  CONTINUE
10.                 PRINT *, X
11.           C
12.           CDIR@ IVDEP
13. Vr------<       DO J1X = 1, 100
14. Vr        C
15. Vr              R1X(J1X) = Y(II(J1X))
16. Vr------>       END DO
17.           CDIR@ IVDEP
18. Vr------<       DO J1X = 1, 100
19. Vr                Y(J1X) = R1X(J1X)
20. Vr------>       END DO
21.                 PRINT *, Y
22.           C
23.           CDIR@ NEXTSCALAR
24. Sr------<       DO 2 I = 2, N-1
25. Sr                Z(I) = Z(I-1) + Z(I + 1)
26. Sr------>    2  CONTINUE
27.                 PRINT *, Z
28.           C
29.           CDIR@ IVDEP
30. Vr------<       DO J1X = 1, 98
31. Vr        C
32. Vr              R2X(J1X) = VX(J1X) + VX(J1X+2)
33. Vr------>       END DO
34.                           CDIR@ IVDEP
35. Vr------<       DO J1X = 1, 98
36. Vr                VX(J1X+1) = R2X(J1X)
37. Vr------>       END DO
38.                 PRINT *, VX
39.                 END
```

```
cft77-8061 cf77: VECTOR SEQ21, Line = 7
  Loop starting at line 7 was not vectorized.  The subscripts
  are ambiguous on "X".
cft77-8004 cf77: VECTOR SEQ21, Line = 13
  Loop starting at line 13 was vectorized.
cft77-8004 cf77: VECTOR SEQ21, Line = 18
  Loop starting at line 18 was vectorized.
cft77-8054 cf77: VECTOR SEQ21, Line = 24
  Loop starting at line 24 was not vectorized.  A NEXTSCALAR
  compiler directive was specified.
cft77-8004 cf77: VECTOR SEQ21, Line = 30
  Loop starting at line 30 was vectorized.
cft77-8004 cf77: VECTOR SEQ21, Line = 35
  Loop starting at line 35 was vectorized.


  1.                      PROGRAM SEQ22
  2.                      DIMENSION X(200)
  3. V------------<       DO 30 I = 1, N
  4. V                      X(I*I) = 1.
  5. V------------>    30 CONTINUE
  6.                      PRINT *, X
  7.                      END

cft77-8004 cf77: VECTOR SEQ22, Line=3
               Loop starting at line 3 was vectorized.

  1.                      PROGRAM SEQ23
  2.                      DIMENSION X(100), Y(100)
  3. V------------<       DO 40 K = 2, 100
  4. V                      X(K) = 2 + Y(K)
  5. V                      Y(K) = COS(X(K))
  6. V------------>    40 CONTINUE
  7.                      PRINT *, Y
  8.                      END

cft77-8004 cf77: VECTOR SEQ23, Line=3
               Loop starting at line 3 was vectorized.
```

```
  1.                      PROGRAM SEQ24
  2.                      DIMENSION VF(500)
  3.                      PARAMETER (R2=100)
  4.                      DEL1 = R2 * 3.1418
  5.                      DELTA = DEL1 - R2
  6. V------------<       DO 1 L = 1, R2
  7. V                      IX = L + DELTA * (2*L-1)
  8. V                      MSUM = MSUM + VF(IX)
  9. V------------>    1  CONTINUE
 10.                      PRINT *, MSUM
 11.                      END

cft77-8004 cf77: VECTOR SEQ24, Line=6
               Loop starting at line 6 was vectorized.

  1.                      PROGRAM SEQ25
  2.                      DIMENSION AM(100,100),PP(100),TAK(100), BM(100),
  3.                     *          VAR(100)
  4.                      PARAMETER (PWR = 3, M = 1, N = 100)
  5.                      GP = 59.45
  6. S------------<       DO 1 I = M, N
  7. S                      VAR(I) = RANF() * 10000
  8. S                      PP(I) = RANF() * 1000
  9. S------------>    1  CONTINUE
 10. S------------<       DO 2 I = M, N
 11. S V----------<         DO 2 J = M, N
 12. S V                      AM(J,1) = PP(I) / TAK (I) ** PWR
 13. S V                      AM(J,2) = SQRT(PP(J))
 14. S V                      BM(J) = VAR(J) * PP(J)-GP*TAK(J) / 144
 15. S-V----------->    2  CONTINUE
 16.                      PRINT *, AM
 17.                      PRINT *, BM
 18.                      END

cft77-8029 cf77: VECTOR SEQ25, Line=6
               Loop starting at line 6 was not vectorized.
               It contains multiple calls to an intrinsic
               function with side effects.
cft77-8035 cf77: VECTOR SEQ25, Line=10
               Loop starting at line 10 was not vectorized.
               It contains an inner loop.
cft77-8004 cf77: VECTOR SEQ25, Line=11
               Loop starting at line 11 was vectorized.
```

```
 1.                      PROGRAM SEQ26
 2.                      DIMENSION Y(100), COEF(100), XM(100,100), YVEC(100),
 3.                     *          RSD(100)
 4.                      PARAMETER (NR = 100)
 5. S------------<       DO 2 I = 1, NR
 6. S                      YO = Y(I)
 7. S                      YV = 0.0
 8. S V----------<         DO 25 J = 1, NR
 9. S V                      YV = YV + COEF(J) * XM(I,J)
10. S V---------->    25 CONTINUE
11. S                     YVEC(I) = YV
12. S                     RS = YV - YO
13. S                     RSD(I) = RS
14. S------------>    2   CONTINUE
15.                       PRINT *, RSD
16.                       END

cft77-8035 cf77: VECTOR SEQ26, Line=5
               Loop starting at line 5 was not vectorized.
               It contains an inner loop.
cft77-8004 cf77: VECTOR SEQ26, Line=8
               Loop starting at line 8 was vectorized.

 1.                       PROGRAM SEQ27
 2.                       REAL G(50, 50), H(50, 50)
 3. S------------<        DO 10 J = 1,50
 4. S S----------<          DO 10 I = 1,50
 5. S S                       G(J,I) = RANF()
 6. S S                       H(J,I) = RANF()
 7. S-S---------->    10 CONTINUE
 8. S------------<        DO 20 J = 1, 49
 9. S Sr---------<          DO 20 I = 1, 49
10. S Sr                      G(I+1,J) = G(I,J) * H(I,J) - H(I,J+1)
11. S-Sr--------->    20 CONTINUE
12.                       PRINT *, G
13.                       END

cft77-8035 cf77: VECTOR SEQ27, Line=3
               Loop starting at line 3 was not vectorized.
               It contains an inner loop.
cft77-8029 cf77: VECTOR SEQ27, Line=4
               Loop starting at line 4 was not vectorized.
               It contains multiple calls to an intrinsic
               function with side effects.
cft77-8035 cf77: VECTOR SEQ27, Line=8
               Loop starting at line 8 was not vectorized.
               It contains an inner loop.
cft77-8044 cf77: VECTOR SEQ27, Line=9
               Loop starting at line 9 was not vectorized.
               It contains a recurrence on G at line 10.
```

FPP translated the previous Fortran code segment as follows:

```
 1.                    PROGRAM SEQ27
 2.              C...Translated by FPP 5.0 (3.03M1) 07/15/91  13:31:44
 3.                    REAL G(50, 50), H(50, 50)
 4.              CDIR@ IVDEP
 5. V----------<       DO J = 1, 2500
 6. V                    G(J,1) = RANF()
 7. V                    H(J,1) = RANF()
 8. V---------->       END DO
 9. S----------<       DO I = 1, 49
10. S           CDIR@    IVDEP
11. S Vs------<          DO J = 1, 49
12. S Vs                   G(1+I,J) = G(I,J)*H(I,J) - H(I,1+J)
13. S Vs------>          END DO
14. S---------->       END DO
15.                    PRINT *, G
16.                    END

                V E C T O R I Z A T I O N   I N F O R M A T I O N
                ------------------------------------------------
cft77-8004 cf77: VECTOR SEQ27, Line = 5
  Loop starting at line 5 was vectorized.
cft77-8035 cf77: VECTOR SEQ27, Line = 9
  Loop starting at line 9 was not vectorized.  It contains an inner loop.
cft77-8003 cf77: VECTOR SEQ27, Line = 11
  Loop starting at line 11 is a short vector loop.
```

```
  1.                     PROGRAM SEQ28
  2.                     DIMENSION A(100,100)
  3. S-----------<       DO 10 I = 1, 100
  4. S Vc---------<        DO 10 J = 1, 100
  5. S Vc                    A(J+4, I*2) = A(J+1,I)
  6. S-Vc--------->    10 CONTINUE
  7.                     PRINT *, A
  8.                     END

cft77-8035 cf77: VECTOR SEQ28, Line=3
               Loop starting at line 3 was not vectorized.
               It contains an inner loop.
cft77-8005 cf77: VECTOR SEQ28, Line=4
               Loop starting at line 4 was vectorized with a computed
               maximum safe vector length.

  1.                     PROGRAM SEQ29
  2.                     DIMENSION B(100,100), C(100)
  3. S-----------<       DO 10 M = 1, 100
  4. S Sr---------<        DO 10 K = 1, 100
  5. S Sr                    B(M*4, K) = B(M*4, K-1) + C(K)
  6. S-Sr--------->    10 CONTINUE
  7.                     PRINT *, B
  8.                     END

cft77-8035 cf77: VECTOR SEQ29, Line=3
               Loop starting at line 3 was not vectorized.
               It contains an inner loop.
cft77-8044 cf77: VECTOR SEQ29, Line=4
               Loop starting at line 4 was not vectorized.
               It contains a recurrence on B at line 5.
```

FPP translated the previous Fortran code segment as follows:

```
 1.                     PROGRAM SEQ29
 2.               C...Translated by FPP 5.0 (3.03M1) 07/15/91  13:42:45
 3.                     DIMENSION B(100,100), C(100)
 4. S-----------<       DO K = 1, 100
 5. S            CDIR@    IVDEP
 6. S Vi--------<         DO M = 1, 100
 7. S Vi                    B(M*4,K) = B(M*4,K-1) + C(K)
 8. S Vi-------->         END DO
 9. S----------->       END DO
10.                     PRINT *, B
11.                     END

 cft77-8035 cf77: VECTOR SEQ29, Line = 4
   Loop starting at line 4 was not vectorized.  It contains an inner loop.
 cft77-8006 cf77: VECTOR SEQ29, Line = 6
   Loop starting at line 6 was vectorized.  An IVDEP compiler directive was specified.
```

```
 1.                      PROGRAM SEQ30
 2.                      DIMENSION D(10,10,10,10), E(10,10)
 3. S------------<       DO 10 I = 1,10
 4. S S----------<         DO 10 J = 1,10
 5. S S                      E(I,J) = RANF()
 6. S S S--------<           DO 10 K = 1,10
 7. S S S Vs-----<             DO 10 L = 1,10
 8. S S S Vs                     D(I,J,K,L) = RANF()
 9. S-S-S-Vs----->    10 CONTINUE
10. S---------------------<        DO 30 II = 1, 10
11. S S-------------------<          DO 30 JJ = 1, 10
12. S S Vs----------------<            DO 30 KK = 1, 10
13. S S Vs W--------------<              DO 30 LL = 1, 5
14. S S Vs W                               D(4,JJ-5,LL,KK) = D(4,JJ-5,LL+1,KK)+
15. S S Vs W                           *                     E(LL,JJ)
16. S-S-Vs-W---------------->     30 CONTINUE
17.                                   PRINT *, D
18.                                   END
```

```
cft77-8035 cf77: VECTOR SEQ30, Line = 3, File = seq30new.f, Line = 3
  Loop starting at line 3 was not vectorized.  It contains an inner loop.
cft77-8035 cf77: VECTOR SEQ30, Line = 4, File = seq30new.f, Line = 4
  Loop starting at line 4 was not vectorized.  It contains an inner loop.
cft77-8035 cf77: VECTOR SEQ30, Line = 6, File = seq30new.f, Line = 6
  Loop starting at line 6 was not vectorized.  It contains an inner loop.
cft77-8003 cf77: VECTOR SEQ30, Line = 7, File = seq30new.f, Line = 7
  Loop starting at line 7 is a short vector loop.
cft77-8035 cf77: VECTOR SEQ30, Line = 10, File = seq30new.f, Line = 10
  Loop starting at line 10 was not vectorized.  It contains an inner loop.
cft77-8035 cf77: VECTOR SEQ30, Line = 11, File = seq30new.f, Line = 11
  Loop starting at line 11 was not vectorized.  It contains an inner loop.
cft77-8003 cf77: VECTOR SEQ30, Line = 12, File = seq30new.f, Line = 12
  Loop starting at line 12 is a short vector loop.
```

FPP Options [A]

When invoked with `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`, FPP, the *dependence analysis* phase of the CF77 compiling system, performs transformations on source code and adds directives to allow vectorization of some loops that might otherwise be difficult to vectorize in the compilation phase.

This appendix describes FPP options that are invoked on the `fpp` command line or, as recommended by CRI, within the `cf77` command in the form `cf77 -Wd"`*fpp_option*`"`.

Command-line options

A.1

`fpp` options are as follows, as used within the `cf77` command.

```
cf77 -Wd" [-C routine1,routine2,...] [-d optoff]
[-D directive[:sub1,sub2,...]] [-e opton] [-F file]
[-I routine1,routine2,...] [-l listingfile] [-M lines]
[-o outputfile] [-p liston] [-q listoff] [-r formaton]
[-n formatoff] [-S file1,file2] [-T threshold] "
```

`-C` *routine1,routine2,...*
: Lists names of concurrently callable routines.

`-d` *optoff* and `-e` *opton*
: Enables (`-e`) or disables (`-d`) optimization option switches named in *optoff* and *opton*. The optimization switches are described in Table 3, page 111.

`-D` *directive*`[:`*sub1,sub2,...*`]`
: Specifies a directive to be applied to certain routines, or to the whole input file if no routines are listed.

`-F` *file*
: Redirects command-line input from the named file. You can use this file to specify `-D` options. An example of a `-F` file follows the description of `fpp` options.

`-I` *routine1,routine2,...*
: Lists routines to be expanded inline.

`-l` *listingfile*
: Enables the FPP listing and directs it to file *listingfile*. This listing shows in detail the actions taken by FPP; it is distinct from the compiler's listings. When you specify `cf77 -Zv -Wf"-em"` or `cf77 -Zp -Wf"-em"`, the resulting listing from the compiler shows the code produced by FPP, rather than your original source. If you specify `cf77 -ZV` or `cf77 -ZP`, FPP's output is saved as *file*`.m`. Any of these methods can be used to show the result of FPP processing.

`-M` *lines*
: Sets the maximum number of lines of code to be allowed for automatic in-line expansion; default is 50.

`-o` *outputfile*
: Directs the translated source to file *outputfile* instead of standard output. The output file is ready for processing by FMP or CFT77, the other components of the CF77 compiling system.

`-p` *liston* and `-q` *listoff*
: Enables (`-p`) or disables (`-q`) listing option switches named in *liston* and *listoff*. The listing switches are described in table 4.

`-r` *formaton* and `-n` *formatoff*
: Enables (`-r`) or disables (`-n`) reformatting (TIDY) option switches named in *formaton* and *formatoff*.

`-S` *file1,file2*
: Specifies file names or complete path names of files (including the actual file name) in which routines to be expanded in-line are located. For example, any of the following specifications is acceptable:

```
-S file.f
-S /usr/fred/file.f
-S /usr/fred/abc.f,xyz.f
```

`-T` *threshold*
: Specifies maximum Autotasking threshold value for comparison to the loop iteration count; default is 800 for CX/CEA systems and 3200 on CRAY-2 systems.

By default, the translated Fortran source output file is written to the standard output file (normally the terminal), and no listing file is produced. If you invoke `fpp` without arguments, it prints a short usage summary.

The following code is an example of a typical `-F` file:

```
-Dnoinner:sub1
-Dnexpand(sub2):sub1#/usr/psr
-Ffile2.com              (Nested command file)
-D relation(n.gt.32):sub2
-Dswitch,tdyoff=p,indal=5,renumb=1000:100
```

To run the Fortran source file `crunch.f` through `fpp`, enter the following:

```
fpp crunch.f > crunch.m
```

The optimized output is sent to `crunch.m`.

To run `crunch.f` through `fpp`, with in-line expansion enabled, and save the output in `crunch.m`; run the output (`crunch.m`) through `fmp`, and save the output in file `crunch.j`; and compile the translated code; enter the following:

```
fpp -e 78 crunch.f > crunch.m
fmp crunch.m > crunch.j
cft77 -a stack -c crunch.j
```

The output of the last command is `crunch.j.o`, which can then be loaded.

`fpp` *optimization and listing switches* A.1.1

Switches, also called option-arguments, let you control optimization of FPP and the contents of the listing file for FPP.

- You can pass optimization switches using `cf77 -Wd"-d"` (disable) and `-Wd"-e"` (enable) or the `SWITCH` directive.
- You can pass listing switches using the `-p` (enable) and `-q` (disable) options of `fpp`, the `-Wd"-p"` and `-Wd"-q"` options of `cf77`, or the `SWITCH` directive.

Table 3, page 111, shows the optimization switches that affect the transformation of the input program. For example, specifying `fpp` option `-d el` means that `EQUIVALENCE` statements are not examined for data dependency analysis, and `IF` loops are not converted to `DO` loops.

Some switches duplicate or overlap the functions of directives. For example, the `-d d` switch is equivalent to the `NODEPCHK` directive with file scope (`CFPP$ NODEPCHK F`).

Switches that correspond to directives (`a`, `c`, `d`, `e`, `i`, `r`, `u`, `v`, and `7`) may be toggled more than once within a routine (using the `SWITCH` directive). Switches that do not correspond to directives (`b`, `h`, `j`, `k`, `l`, `m`, `o`, `p`, `s`, `t`, `y`, `0`, `1`, `4`, `5`, and `6`) can have only one valid setting for any one routine; if they are set more than once within a routine, only the last setting is used.

The `q`, `x`, and `8` switches are valid only as command line option-arguments.

Table 4, page 114, shows the switches that control the format of the listing file. For example, if you wanted to get a 132-column printer listing without warning messages and no event summary, you would specify `cf77 -Wd"-q twe"`.

The TIDY subprocessor is a feature of FPP that improves the readability of the output code, either by using default standards, or according to user-specified parameters. By default, TIDY is applied only to loops that require restructuring in order to vectorize and concurrentize. To apply TIDY to the entire program unit, use `cf77 -Wd"-dy"`.

Table 3. Optimization switches enabled and disabled by `-e` and `-d`

Switch	Description	Default
a	Allows associative transformations. `-d a` is equivalent to the `NOASSOC` directive with file scope.	ON
b	Generates linear recursion library calls.	OFF
c	Autotasks loops; all loops (inner and outer) that have enough work to justify concurrent execution are analyzed for Autotasking. `-d c` is equivalent to the `NOCONCUR` directive with file scope.	ON
d	Does not ignore potential data dependencies. `-d d` is equivalent to the `NODEPCHK` directive with file scope.	ON
e	Examines `EQUIVALENCE` statements for data dependency. `-d e` is equivalent to the `NOEQVCHK` directive with file scope.	ON
f	Enables source line debugging.	OFF
h	Allows parallel case optimization. Ignored if the `c` switch (autotask) is off.	ON
i	Analyzes inner loops with variable iteration counts at compile time to determine if they are candidates for Autotasking. By default, outer loops and inner loops that obviously have enough work are autotasked. For inner loops with high iteration counts and many statements, enabling this option may improve performance. `-e i` is equivalent to the `INNER` directive with file scope. The `i` switch is ignored if the `c` switch (autotask) is off.	OFF
j	Translates nested loop idioms, such as matrix multiplication matrix-vector multiplication, and rank one update, to library calls.	ON
k	Treats `D` in column 1 as a comment character. If this switch is off, a `D` in column one is treated as a blank. This switch provides compatibility with a debugging feature of some compilers.	ON
l	Transforms `IF` loops to `DO` loops.	ON
m	Generates alternative code for potential dependencies. If this switch is off, loops containing potential data dependencies will not be optimized.	ON
o	Specifies minimum `DO` trip count is one. Provides compatibility with ANSI 66 Fortran compilers.	OFF

Table 3. Optimization switches enabled and disabled by -e and -d (continued)

Switch	Description	Default
p	Collapse loop nests into single loops when possible.	ON
q	Takes error exit if syntax or fatal errors are found. If this switch is on and fpp detects a syntax or fatal error, it returns an error code of 2. If fpp was invoked by cf77, cf77 ceases processing at that point.	OFF
r	Splits out all user subroutines and functions. -e r is equivalent to the SPLIT directive with file scope.	OFF
s	Permits loop splitting to isolate recursion, which permits partial vectorization of loops.	ON
t	Specifies use of aggressive loop exchange criteria. Gives greater weight to the use of stride-one vectors and increased vector length, compared to retaining original loop nest ordering.	OFF
u	Generates final values for transformed scalars when appropriate. -d u is equivalent to the NOLSTVAL directive with file scope.	ON
v	Enhances CFT77 vectorization. -d v is equivalent to the NOVECTOR directive with file scope. If this switch is off, the b, m, p, r, and s switches are not meaningful.	ON
x	Creates optimized source file. This switch may be turned off if only the diagnostic listing is wanted. Turning this switch off may speed compile time and reduce disk space used. The setting of this switch does not affect the listing of the transformed source in the listing file. This switch is valid only as a command line option-argument; it may not be specified with the SWITCH directive.	ON
y	Reformats only restructured loops. -d y causes the entire program unit to be reformatted with the TIDY subprocessor.	ON
0	Generates Autotasking threshold test.	ON
1	Converts array syntax to DO loops.	ON
3	Enables 80-column input.	OFF
4	Asserts that first values of private arrays are not needed. In some cases, allows more loops to be autotasked.	OFF

FPP directives
A.2

FPP directives are lines appearing in source code beginning with the string `CFPP$`. You can use them to give FPP more information about your program. (These directives are also called *user directives*.)

Compiler directives are source lines beginning with `CDIR$` and `CDIR@` that are interpreted by the compiler as information about the program. FPP automatically inserts `CDIR@` directives. FPP also interprets certain compiler directives associated with vector processing. (`CDIR$` directives are the same directives used by the CFT77 compiler.)

Note

Directives beginning `CDIR@` should not be inserted by users.

FPP user directives have the following syntax:

```
CFPP$ directive scope
```

The `C` in column 1 makes the directive a comment for all other Fortran compilers. The body of the directive begins after one or more blanks. Following the directive is an optional scope parameter, *scope*, which specifies the range of code to which the directive applies. Table 5 shows allowable *scope* values.

Table 5. Allowable scope parameters for `CFPP$` directives

Value	Meaning	Description
`R`	Routine	Directive applies until the end of the current routine.
`L`	Loop	Directive applies to the next loop encountered. Applies only to `DO` loops.
`F`	File	Directive applies until the end of the input file.
`I`	Immediate	Directive applies immediately at that point in the source code.
	Blank	Same as `L`; directive applies to the next loop encountered.

Some directives ignore the scope parameter. Directives affecting IF loops must have R or F scope. Many directives can be preceded by NO, thus effecting the reverse operation.

The following example tells FPP to ignore potential data dependencies in the next loop:

```
CFPP$ NODEPCHK
```

The following example turns off the vectorization enhancement for the rest of this routine.

```
CFPP$ NOVECTOR R
```

The following example turns on the listing for rest of file.

```
CFPP$ LIST F
```

The full set of directives is summarized in Table 6. The scope entry is either L, indicating that it applies to the next loop; R, indicating that it applies to the whole routine; I, indicating that it applies immediately; or LRF, which indicates that any of the loop, routine, or file options can be used to control the scope. If the scope is not specified, the default is L, or loop. A short description of each of these directives follows the table. The default condition is indicated by an asterisk (*).

Table 6. FPP directives

Directive	Function	Scope
SWITCH, *w=s,w=s...*	Sets global switches. Switch *s* is passed to keyword *w*. Keywords correspond to fpp options as follows: OPTON, -e; OPTOFF, -d; LSTON, -p; LSTOFF, -q; TDYON, -r; TDYOFF, -n.	I
NOVECTOR/VECTOR*	Disables/enables vectorization enhancement. See v option.	LRF
NOCONCUR/CONCUR*	Disables/enables Autotasking; does not affect vectorization. See c option.	LRF
SKIP	Disables Autotasking and vectorization.	LRF
INNER/NOINNER*	Allows/disallows Autotasking for inner loops.	LRF
CNCALL	Allows concurrent calls in loop.	LRF

Table 6. FPP directives (continued)

Directive	Function	Scope
CHOP_HERE	Chops a loop into two loops at this point.	None
NOALTCODE/ALTCODE[*n*]*	Does not generate/generates alternate code blocks. *n* is an integer indicating a trip count or any other expression to be used in tests for alternate code. See option m.	LRF
NOASSOC/ASSOC*	Does not/does perform associative transformations.	LRF
SPLIT/NOSPLIT*	Does/does not enable cutting subroutine and function calls from loop (tells FPP that one loop pass does not feed results to the next).	LRF
SELECT	Selects next loop to optimize in a nest of loops (overrides FPP's choice; directive ignored if FPP finds problem with loop).	L
NOLSTVAL/LSTVAL*	Does not/does save last values of transformed scalars (indexes or promoted scalars; especially those in common or used in array subscripts) within directive's scope. NOLSTVAL says that values do not need to be same as in scalar version (for later use).	LRF
UNROLL[*n*]/NOUNROLL[*n*]*	Enables/disables loop unrolling. With scope R or F, *n* specifies maximum trip count for automatic unrolling (default 3). With L scope, *n* is the number of times to unroll loop (default calculated by FPP).	LRF
NODEPCHK/DEPCHK*	Ignores potential data dependencies or performs dependency check; asserts that no recursion exists. See option d.	LRF
NOSYNC/SYNC*	Does/does not ignore potential overlap of array elements accessed by different processors.	LRF
NOEQVCHK/EQVCHK*	Does not/does check EQUIVALENCE statements to see if they cause data dependencies. See e option.	LRF
PERMUTATION (*ia1*, *ia2*...)	Declares that listed integer arrays, for use as subscripts in array section names, have no repeated values.	R
RELATION (*int1 rel int2*)	Specifies relation between an integer variable and another integer value. *rel* is a Fortran relational operator.	LRF

Table 6. FPP directives (continued)

Directive	Function	Scope
NOLIST/LIST*	Turns off/on FPP listing; see option `ql`, page 111.	I
ITERATIONS (*v1=n1,v2=n2,...*)	Like COUNT but supplies values, *n*, for loop indexes by variable name, *v*.	R
EXPAND [(*r1,r2...*)]	Expands listed routines, *r*, or all routines in-line; independent of CFT77 optimization. See option `-I`, page 107.	n/a
NEXPAND (*list*) (*#path*)	Expand routines in *list* found in directory *path*, as well as routines called by them. Nested routines are not expanded by any other means.	n/a
SEARCH (*files*)	Specifies files to be searched for routines that are expanded in-line. Default file is *rtn*`.f` for routine *rtn*.	n/a
COUNT (*i*)	Supplies approximate iteration count *n* for loops within the directive's scope, affecting whether loops are autotasked.	LRF
AUTOEXPAND/ NOAUTOEXPAND*	Enables/disables automatic routine inlining.	LRF

Transformation directives A.2.1

Transformation directives change the way FPP transforms a loop.

NOVECTOR/VECTOR A.2.1.1

NOVECTOR disables vectorization enhancement. Despite the best efforts of FPP to make the right choices, occasionally a loop may be less efficient after transformation. NOVECTOR is provided to disable vectorization enhancement in such cases. VECTOR only toggles back from NOVECTOR; it does not force vectorization.

The `cf77 -Wd"-d v"` option is equivalent to NOVECTOR with file scope.

SKIP
A.2.1.2

SKIP disables Autotasking and vectorization; it acts like a combined NOCONCUR and NOVECTOR.

CHOP_HERE
A.2.1.3

CHOP_HERE directs FPP to split a loop at the indicated line. This directive is used for fine-tuning certain loops.

Example:

```
      SUBROUTINE TEST
      COMMON N, A(500), B(500)
      ...
      DO 10 I = 1, N
         IF ( A(I) .GT. 0 ) THEN
            S = SQRT(A(I))
CFPP$ CHOP_HERE
            B(I) = S + 1.0/S
         ENDIF
 10   CONTINUE
```

Translation:

```
      TASK COMMON/Z1FPP0CM/ QQQ(8191)
      EQUIVALENCE (QQQ(1),S1U), (QQQ(506),L1V)
      ...
CDIR@ IVDEP
      DO 10 I = 1, N
         L1V(I) = A(I) .GT. 0
         IF (L1V(I)) S1U(I) = SQRT(A(I))
   10 CONTINUE
CDIR@ IVDEP
      DO 77001 I = 1, N
         IF (L1V(I)) B(I) = S1U(I) + 1.0/S1U(I)
77001 CONTINUE
```

ALTCODE/NOALTCODE
A.2.1.4

For potentially dependent vector loops, ALTCODE directs FPP to generate versions of the loop both with and without an IVDEP directive. ALTCODE also directs FPP to generate a run-time test to choose between them based on the value of array subscript expressions.

For autotasked loops, ALTCODE directs FPP to supply a threshold test for the IF clause of the DO ALL or DO PARALLEL.

ALTCODE allows an optional parameter. If the parameter is an integer constant, FPP generates a test comparing the loop's iteration count to the constant. If the parameter is not an integer constant, the parameter is echoed verbatim for the IF test.

Note that ALTCODE is on by default. The -d m option to fpp is equivalent to NOALTCODE with file scope.

NOASSOC/ASSOC
A.2.1.5

By default, FPP transforms certain constructs into vector or concurrent versions in which the order of operations may be different than the original (that is, they have been associatively transformed). (This is like the associative property of real numbers.) Because of the way numbers are internally represented in computers, this operation reordering may result in answers that differ slightly from the scalar original. For example, floating-point arithmetic is not associative. The NOASSOC directive disables all associative transformations, including the following:

- Reductions – sum, dot product, and index of minimum and maximum.
- Operation reordering when minimizing dependent regions.
- Linear recursion translation.

The cf77 -Wd"-da" option is equivalent to NOASSOC with file scope.

SPLIT/NOSPLIT
A.2.1.6

SPLIT asserts that subroutine and function calls do not have side effects that cause feedback of results from one loop pass to another, and thus may be "split out" from an optimized loop into a separate loop.

SELECT
A.2.1.7

SELECT advises FPP to choose the next loop as the one to vectorize or autotask in a nest of loops. If FPP cannot analyze the loop or finds a dependence, the SELECT directive is ignored.

In choosing a single loop from a nest, FPP weighs loop iteration count, the presence of data dependence, and the amount of work within the loop to a heuristic algorithm. Because not all pertinent information is available at compile time, FPP may not always be able to make the best choice. Therefore, the SELECT

directive allows you to dictate the optimization mode of a specific loop. Place the SELECT directive directly before the DO statement of the loop to be optimized. An optional argument indicates the mode of optimization, either VECTOR or CONCUR. The default is VECTOR.

NOLSTVAL/LSTVAL
A.2.1.8

NOLSTVAL advises FPP that final values for transformed scalars (which are either indexes or promoted scalars; see page 128). This directive is useful when FPP cannot determine by inspecting the current subprogram whether or not a variable is subsequently used. Such variables are typically in COMMON or are array references. Unrolling is discussed on page 85.

NOUNROLL/UNROLL
A.2.1.9

The UNROLL directive has two functions: the first function is to enable/disable automatic unrolling of loops with small constant iteration counts; the second function is to force explicit unrolling of a particular loop, regardless of iteration count. Eliminating an inner loop by unrolling may allow another loop to vectorize.

These directives have the following syntax:

```
CFPP$  UNROLL   [(number_of_times)]   [{L,R,F}]
CFPP$  NOUNROLL  [{L,R,F}]
```

When routine or file scope is specified (R or F), automatic unrolling of loops is enabled or disabled over that scope. The optional parameter *number_of_times*, which must be a constant, specifies the threshold loop iteration count for automatic unrolling. Loops with an iteration count greater than this value are not unrolled. If no parameter is specified, the threshold is 3.

To force a loop to be explicitly unrolled, use the UNROLL directive with local scope (L) immediately preceding the loop. In this case, the optional parameter is taken as the number of times to unroll the loop. If a parameter is not supplied, FPP uses an internally calculated function of the loop length, loop complexity, and default threshold.

Data dependency directives A.2.2

Data dependency directives are used to help FPP decide whether data dependency conflicts actually exist in a loop. If you know that an operation is not recursive, you can supply one of these directives to inform FPP. These directives are discussed further and examples are given in "Using data dependency directives," page 56.

NODEPCHK/DEPCHK A.2.2.1

When elements of an array are modified within a loop, FPP must determine the exact storage relationship of these elements to all other references to the array in the loop. This must be done to ensure that the references do not overlap, and thus can be safely executed in parallel. When the relationships cannot be determined, FPP issues a *potential dependency* diagnostic to the listing file.

The NODEPCHK directive asserts that all such potentially recursive relationships are, in fact, not recursive. You should use this capability only when you know no real recursion exists. Use of the directive does not, however, force the optimization of operations that are unambiguously recursive. The DEPCHK directive is used to toggle back to the default state. The `cf77 -Wd"-d d"` option is equivalent to NODEPCHK with file scope.

NOEQVCHK/EQVCHK A.2.2.2

NOEQVCHK directs FPP to ignore relationships between variables caused by EQUIVALENCE statements, when examining the data dependencies in a loop. The `cf77 -Wd"-d e"` option is equivalent to NOEQVCHK with file scope.

PERMUTATION A.2.2.3

PERMUTATION declares that an integer array does not have repeated values. This is useful when the integer array is used as a subscript for another array (*indirect addressing*). If it is known that the integer array is used merely to permute the elements of the subscripted array, it can often be determined that feedback does not exist with that array reference.

RELATION A.2.2.4

RELATION advises FPP that a specified relationship exists between two integer variables or between an integer variable and an integer constant. This information may be useful to FPP in resolving otherwise ambiguous array relationships.

RELATION directives are *informative only*, they do *not* force any action. They can be applied at the loop, routine, or file level. If conflicting relations are given, the result is unpredictable. It is up to you to ensure that the relations specified are correct and consistent. See page 59 for more information.

Advisory directives
A.2.3

Advisory directives provide information for FPP, which may result in a better choice of loops to be optimized.

COUNT and ITERATIONS
A.2.3.1

If the iteration count of a loop (or class of loops) is variable and cannot be determined from the information in the routine until execution time, but you know the approximate number of iterations, you can use the COUNT or ITERATIONS directive to supply this information.

```
CFPP$  COUNT (val1) [{L,R,F}]

CFPP$  ITERATIONS (var1=val1 [,var2=val2] ...)
```

val1, val2... Specify vector length values. These values do not have to be exact because they are used only as guidelines.

var1, var2... Specifies indexes of a loop with the given assumed vector length values.

The COUNT directive can be used at the file, routine, or loop levels. The ITERATIONS directive can only be used at the routine level. A CFPP$ COUNT(0) F or NOITERATIONS directive returns FPP to its normal iteration count processing.

Example:

```
      SUBROUTINE OPTIM7 ( A, B, C, L, M, N )
      REAL A(L,M,N), B(L,M,N), C(M,N), S
C
CFPP$ ITERATIONS ( I=2, J=100, K=100 )
      DO 609 K = 1,N
         DO 608 J = 1,M
            C(M,N) = S + C(M,N)
            DO 607 I = 1,L
               A(I,J,K) = B(I,J,K) + C(M,N)
  607       CONTINUE
  608    CONTINUE
  609 CONTINUE
```

By default, the inner loop would be vectorized, but because the directive indicates that the iteration count is small, and neither of the outer loops is a good candidate for translation, the entire loop nest is left alone.

Listing directives
A.2.4

Listing directives change the appearance of the FPP listing. The following subsections discuss the FPP listing directives.

NOLIST/LIST
A.2.4.1

You can selectively suppress listing of the input source with the NOLIST/LIST directive pair. If NOLIST (or the `cf77 -Wd"-ql"` option) is in force when the END statement is encountered, the rest of the listing (messages, translated source, summaries) is suppressed, unless specifically enabled (with the `cf77 -Wd"-p"` option switches.

SWITCH
A.2.4.2

SWITCH enables you to set (or change) global switches, including listing switches. You can also use the SWITCH directive to set optimization and reformatting switches. See Table 3, page 111, for a list of the optimization switches. See Table 4, page 114, for a list of listing switches.

The format of the SWITCH directive is as follows:

```
CFPP$ SWITCH,OPTON=str,OPTOFF=str,LSTON=str,
LSTOFF=str,TDYON=str,TDYOFF=str,SPACE=integer
```

Keywords OPTON, OPTOFF, LSTON, LSTOFF, TDYON and TDYOFF correspond to fpp options -e, -d, -p, -q, -r, and -n, respectively.

The SPACE parameter lets you specify a size, in words, for the FPP-generated common block. The default size is 8191 words.

Blanks are not significant, and keywords and switches can be either uppercase or lowercase.

In-line expansion directives
A.2.5

In-line expansion directives provide information for FPP that allows expansion of the bodies of certain subroutines and functions into the loops that call them. The directives are as follows:

- AUTOEXPAND
- EXPAND
- NEXPAND
- SEARCH

See "In-line expansion," page 131, for more information about these directives and examples of their use.

Additional Source Transformations [B]

Additional Source Transformations [B]

When the `cf77 -Zv` option is invoked, the compiling system performs source transformations for scalar optimization, replacement of library calls, and (if specified) in-line expansion. These forms of processing are in addition to transformations already described in this manual.

Library calls
B.1

CF77 recognizes certain Fortran coding sequences that it can replace by calls to highly optimized library routines, as follows:

- Matrix multiplication in most common forms
- Certain linear recursion patterns
- Index of maximum and minimum element operations

Translation of linear recurrence
B.1.1

CF77 recognizes and translates certain forms of first- and second-order linear recurrence into *libsci* calls. Calls to routines `FOLR`, `FOLRP`, `SOLR`, and `SOLR3` are generated.

Example:

```
      DO 75 I = M, N
 75     B(I) = B(I)-B(I-1) + A(I)
```

Translation (when compiled with `cf77 -Zv`, `-ZV`, `-Zp`, or `-ZP`):

```
      CALL FOLR (N-M+2,A(M-1),1,B(M-1),1)
```

Linear recurrence translation can be enabled by `-Wd"-eb"`. If the `NOASSOC` directive or `-Wd"-da"` is in force, linear recurrence is not translated.

Scalars in loops
B.2

Scalar variables are single locations in memory, such as a simple variable `X`. Array references whose subscript values are invariant in a loop (and thus represent a single location through all passes of the loop) are called *array constants*. CF77 treats array constants similarly to simple scalar variables.

In general, scalar variables must be defined (appear on the left side of the equal sign) before they are used (appear on the right side of the equal sign) if they are defined in the loop. Otherwise, they are *carry-around scalars*, which may inhibit optimization.

In addition, scalar variables that are modified in a loop can sometimes inhibit optimization. This subsection discusses the transformations used to deal with the storing of values into non-indexed scalars within a loop.

Scalar promotion
B.2.1

When a scalar is set to a vector expression, and then used in more than one part of a split loop, that scalar must be *promoted* to a vector. This requires the introduction of temporary vectors that replace the promoted scalars.

Last values of promoted scalars
B.2.1.1

When CF77 restructures a loop, it may eliminate the definition of some scalar temporaries. When necessary, it re-creates the correct final values of such variables. (See the previous discussion of last value saving.)

Conditionally defined promoted scalars
B.2.1.2

If the final value of a conditionally defined promoted scalar is required, that scalar cannot be optimized, because there is no efficient way to determine the "last true" element. This may inhibit Autotasking and degrade vectorization enhancement.

Example:

```
      SUBROUTINE CND2
      COMMON /SCAL/ S,N
      COMMON /ARRY/ A(1000),C(1000)
C
CFPP$ INNER
      DO 30 I = 1, N
        IF(A(I) .GT. 0.) THEN
          S=1./A(I)
          C(I) = S + C(I)
        ENDIF
30    CONTINUE
      RETURN
      END
```

In this example, the fact that S is in COMMON, and defined conditionally in the loop, prevents FPP from Autotasking the loop. If the final value of S is not needed (as is probably the case), you can use the NOLSTVAL directive, or the `cf77 -Wd"-du"` option, to allow optimization. An even better solution would be to make S a local variable, rather than a COMMON variable.

Carry-around scalars B.2.2

Scalars that may be used before they are defined in a loop are called *carry-around* scalars. They may or may not be recursive. Recursive carry-around scalars inhibit optimization. All references to these variables are collected in a scalar loop and split out from the rest of the calculation if possible.

Example:

```
      DO 3313 I = 1,N       ! Not optimized
      A(I) = S + 1/S        ! S is "carried around"
      B(I) = C(I) - A(I) + S
3313  S = B(I) + D(I)
```

Equivalenced scalars B.2.3

In some cases, scalars that are equivalenced may inhibit data dependency analysis. The NOEQVCHK directive may be used to allow such operations to optimize, if the equivalencing does not actually create recursion. (It almost never does. See section 3

for a full discussion of data dependence and the NOEQVCHK directive.)

Example:

```
      COMMON /BLOCK/ A(999)
      EQUIVALENCE (S,A(100))
      .
      .
      DO 67 I = M, N          ! Not optimized
         S = B(I)**2
         A(I) = S + 1.0/S
67    CONTINUE
```

ASSOC *directive* B.2.4

A *reduction function* is an operation that condenses array operands into one scalar value that characterizes some aspect of the input arrays.

FPP ensures that the optimized code is semantically equivalent; however, FPP may reorder code and the order of operations grouped from that in the original scalar loop. In some very sensitive calculations, this reordering can cause small numerical differences; for example, altering the result in the final decimal places, because of round-off differences.

The optimized implementation of a reduction function often combines operand elements in an order that may differ from that specified in the original scalar loop. In some very sensitive calculations, this reordering can alter the result in the final decimal places, due to roundoff differences.

The ASSOC and NOASSOC directives let you disable or enable all transformations that change the way operands are associated with each other. You can also enable or disable associative transformations globally by using `cf77 -Wd"-ea"`. (An example of an associative transformation is changing `(X+Y)+Z` to `X+(Y+Z)`.) Such a transformation is always mathematically correct, but it does not always produce exactly the same result on computers, because they use finite-sized words to represent numbers.

The most common type of associative transformation made by CF77 is the generation of autotasked reductions. If the NOASSOC directive is specified, these optimizations are not done; otherwise, by default, associative transformations are performed. Another example of associative transformation occurs when CF77 splits loops to minimize recursion.

In-line expansion

B.3

This subsection describes CF77's source-level in-line routine expansion; this is distinct from that performed in the compiling phase. In-line expansion reduces subroutine calling overhead and increases vectorization possibilities. It also allows scalar optimizations such as invariant code relocation and common subexpression analysis to extend across the body of the subroutine, as well as the body of the calling loop.

There are two modes of inlining: automatic and explicit. These modes can be requested by directive or by command-line or control statement specification. Application codes sometimes have small external functions that are called from inside many loops; these functions are good candidates for in-line expansion.

The following is a small example of in-line expansion:

Original:

```
      DO 100 I = 1, N
         A(I) = CALC (A(I), X+B(I), 2.0)
100   CONTINUE
      ...
      END
      FUNCTION CALC (A,B,C)
      CALC = A + SQRT( B**2 + C**2 )
      IF (CALC.LT.0) CALC = ABS (B + C)
      END
```

Expanding function CALC inline:

```
      DO 100 I = 1, N
         TEMP1X = X + B(I)
         CALC1X = A(I) + SQRT(TEMP1X**2 + 2.0**2)
         IF(CALC1X.LT.0) CALC1X = ABS(TEMP1X+2.0)
         A(I) = CALC1X
100   CONTINUE
```

Automatic in-line expansion
B.3.1

Automatic inlining, specified on the command line, gets rid of "leaves" of the calling tree.

When automatic inlining is enabled (using the `AUTOEXPAND` directive or `cf77 -Wd"-e7"`) CF77 expands every called subroutine or function that meets the following criteria:

- It has less than a threshold number of lines of code. The default is 50, but the you can change the threshold with `cf77 -Wd"-M"`.
- It contains no calls inside the expanded routine (no nesting is allowed with automatic expansion).
- It contains no expansion inhibitors (for example, argument types must match).

Full versus safe expansion
B.3.1.1

Two forms of automatic in-line expansion can be used: full and safe. Full automatic in-line expansion is enabled by `cf77 -Wd"-e7"`. Safe automatic in-line expansion is enabled by `cf77 -Wd"-e6"`.

Safe mode is provided because in certain unusual circumstances FPP cannot correctly determine how to inline occurrences of dummy arrays whose dimensions differ from those of the actual arrays that were passed to them. In safe mode, these are not inlined; whereas in full mode, they are inlined. In either case, a warning is put in the listing file. The following example shows a case where full mode inlining produces incorrect code.

Example:

```
      SUBROUTINE SAM(A,B,LDA,N)
      REAL A(LDA,*), B(*)
      DO 886 I = 1, N
         B(I) = ADD(N, A(1,I), 1)
886   CONTINUE
      END

      REAL FUNCTION ADD(N, X, INCX)
      REAL X(2:1+INCX,1)
      Y = 0
      DO 882 J = 1, N
         Y = Y + X(1,J)
882   CONTINUE
      ADD = Y
      END
```

In full mode, FPP inlines ADD into SUM, with the following diagnostic messages and output:

```
fpp-345 fpp: WARNING SAM, Line = 4, File = safe.f
   Possible discrepancy between actual and dummy
    array extents.
fpp-222 fpp: WARNING SAM, Line = 4, File = safe.f
   Routine 'ADD' expanded.
```

Output produced:

```
CMIC@ DO ALL IF (N .GT. 20) SHARED(N, LDA, A, B)
CMIC@1   PRIVATE(Y, I, ADD1X, J)
      DO 886 I = 1, N
C*****  Code Expanded From Routine:  ADD
      Y = 0
CDIR@ IVDEP
      DO J = 1, N
         Y = Y + A(1,J+I-1)
      END DO
      ADD1X = Y
      B(I) = ADD1X
C*****  End of Code Expanded From Routine:  ADD
 886  CONTINUE
```

The J loop addresses A along the second dimension, striding by LDA, when it should be striding by one.

In safe mode, FPP does not inline ADD, but it does generate the following diagnostic message:

```
fpp-343 fpp: WARNING SAM, Line = 4, File = safe.f
   Extent of dummy and/or actual array cannot be
   determined.
```

Two options are provided because in safe mode, FPP sometimes detects possible problems when there is no actual problem. In practice, FPP rarely detects a problem; generally, either mode produces similar results. Use safe mode if you suspect a dimensioning conflict of the kind previously described.

Explicit in-line expansion
B.3.1.2

In explicit in-line expansion, you list the routines to be expanded, or direct expansion in the following line of code by using the EXPAND directive. You can list routines for explicit in-line expansion on the command line when CF77 is invoked using `cf77 -Wd"-I"` or the EXPAND directive.

You can request nested expansion with explicit inlining. The NEXPAND directive expands the indicated routine and all routines it calls, leaving no external references.

In-line expansion directives
B.3.2

This subsection describes the expansion directives used to control source-level in-line expansion.

AUTOEXPAND *directive*
B.3.2.1

AUTOEXPAND invokes automatic routine expansion. It has the following syntax:

```
CFPP$ AUTOEXPAND
```

The AUTOEXPAND directive with global scope is equivalent to `cf77 -Wd"-e7"`.

EXPAND *directive*
B.3.2.2

EXPAND provides explicit routine expansion and is equivalent to `cf77 -Wd"-I`*arg*`"` . It has the following syntax:

```
CFPP$  EXPAND [ ( list ) ]
```

list is a list of routines to expand within this routine; if it is not supplied, any calls to the next statement are expanded. Scope is ignored on the EXPAND directive.

Example:

```
CFPP$ EXPAND (CALC)
      ...
      DO 100 I = 1, N
         A(I) = CALC( A(I), B(I)+1., N )
 100  CONTINUE
      ...
      END
```

NEXPAND *directive*
B.3.2.3

Nested expansion is done only if specified by user directives. Nested routines are not expanded in auto expansion mode, thus reducing the possibility of greatly expanding code size. You must explicitly specify nested routines in order for them to be expanded.

The NEXPAND directive specifies explicit nested routine expansion; it has no command-line equivalent. NEXPAND specifies a list of routines; these routines are to be expanded, together with all routines that they call directly or indirectly. It has the following syntax:

```
CFPP$  NEXPAND [(list) [#path] ]
```

list is a list of routines to expand. *path* specifies the path where the routines to be expanded are located. Scope is ignored on the NEXPAND directive.

Example:

```
CFPP$ NEXPAND ( CALC )#/USR/TEST
      SUBROUTINE ADD
           .
           .
      DO 100 I = 1, N
         A(I) = CALC( A(I), B(I)+1., N )
 100  CONTINUE
      ...
      END
```

Here CALC would be expanded and any calls inside CALC would also be expanded. CF77 looks for `calc.f` in path `/usr/test`.

Where to get the code
B.3.3

To automatically expand a routine, CF77 needs to know where to find the routine's source code. The source location depends on programming style and on the user interface.

SEARCH *directive*
B.3.3.1

SEARCH tells CF77 where to look for the routines to expand. It is equivalent to `cf77 -Wd"-S"` and has the following syntax:

```
CFPP$  SEARCH (files)
```

files is a list of files (separated by commas) in which to search for the routines to be expanded. If *files* is the special entry `*.f`, routine `xyz` is looked for in file `xyz.f`. (This is the default search method.)

Same file
B.3.3.2

Called routines can be searched for in the same file as the calling routine (*stacked input*). This requires an initial pass by CF77 through the entire input file (and files read in by `INCLUDE` statements) to build a directory of the program units in the input file. `cf77 -Wd"-e8"` enables this initial pass, which by default is not done.

Explicitly named file
B.3.3.3

Using the `SEARCH` directive or `cf77 -Wd"-S"`, you can supply the name of a file in which to search for called routines.

Implicitly named file
B.3.3.4

Fortran programs are frequently stored so that each routine of the program resides in a separate file with a canonical name (for example, the name of the routine followed by ".f"). The `.f` files are searched for in the current working directory. This is the default search method when inlining is requested and the `-e8` option is not specified.

Possible problems
B.3.4

The following subsections discuss possible problems you may encounter when using in-line expansion.

Separate compilation
B.3.4.1

Fortran allows program units to be compiled separately and linked together. Because different program units can be compiled at different times, in-line expansion is not foolproof.

For example, suppose program `A` has subroutine `B`, which calls subroutine `C`. Subroutine `C` is expanded into subroutine `B`. Later, we decide to change the calculation in subroutine `C` and so we edit it; rather than recompiling the entire program, we just recompile `C` and link it with the previously compiled

routines. Our changes to `C` are not present in the actual code executed because `C` is no longer called from `B` (it was expanded.)

Thus, you must be involved in the routine expansion process at least to the point of knowing which routines must be recompiled when a change is made. For this reason, CF77 generates both a warning message for every expansion and an expansion event table so that it is clear what has been expanded. (You may want to use the `make` command to build your application. Using `make` lets you explicitly record these kinds of dependencies, so that proper recompilation is performed.)

Code size
B.3.4.2

A problem that may result from in-line expansion is code mushrooming; if too many routines are expanded, or the expanded routines are too large, the size of the compiled code may reach unacceptable proportions. This potential problem can be controlled through judicious application of this transformation.

Debugging
B.3.4.3

In-line code expansion can complicate user run-time debugging; if the program fails in an expanded section of code, the error is reported in a different routine than the one in which it originally appeared. Further, referring to the in-lined routine by name (such as in a breakpoint) will not find any references or will find the non-expanded calls to the routine.

CF77 records (in a comment line in the listing file) the original line number of the routine invocation on all the expanded lines.

Compilation rate
B.3.4.4

Any scheme to analyze more than one program unit at a time leads to significantly slower compilation rates, and in-line expansion is no exception. It may result in two passes over the entire program, and potentially much longer compile times, depending on how much code is inlined.

Analysis inhibitors
B.3.5

There are several ways in which analysis may be inhibited, thereby prohibiting expansion. An appropriate message in the listing informs you of a failed expansion. A full list of messages appears at the end of this section.

Expansion inhibitors
B.3.5.1

The following are expansion inhibitors:

- The routine to be expanded cannot be located.
- Syntax errors are found in the expansion routine.
- The arguments used in the calling sequence do not match the arguments in the expansion routine.
- There is a conflict between common blocks of the calling routine and the expansion routine.
- A function name referenced in the expansion routine conflicts with a nonfunction name used in the calling routine.

Automatic expansion inhibitors
B.3.5.2

In automatic mode, all calls to routines must meet the following criteria to be expanded:

- The routine to be expanded has less than the maximum allowed number of noncomment lines. The default is 50 noncomment lines. The default can be changed by using `cf77 -Wd"-M".`
- The routine to be expanded does not call any other external routines.
- There are no inhibitors to the expansion (common blocks that do not agree, and so on.)

If these criteria prove too restrictive, you can always resort to explicit mode. The objective of automatic mode is to catch small, simple external functions that could have been statement functions. More demanding cases must be explicitly requested.

Although called automatic, this mode must be enabled by `cf77 -Wd"-e6"` or `-Wd"-e7"`, or by the `AUTOEXPAND` directive. An informational message is issued for each expansion action. This is to remind you that expanded routines need to be recompiled each time they are changed.

In-line expansion mechanics
B.3.6

CF77 renames all variables and parameters in expanded program units by combining the first four characters of the variable with an integer number and the suffix `X` to create a unique name. For example, the local variable `I` could become

I1X, and the next reference to I in another expanded routine could become I2X.

Operations that are performed during in-line expansion of subroutines and functions are as follows:

- Common blocks: All common blocks that are used in both routines agree. The callee's version of the common block may be a subset of the caller's. Elements of the common block may have different names in caller and callee, but must not differ in size or data type.
- Constant parameters (from PARAMETER statements): Names are examined and changed if necessary to avoid conflict with any name from the calling program. If identical in name and value with a constant parameter in the calling program, then no change is made.
- Local variables: Local variables used in the expanded routine are checked against all names defined in the caller, and made unique.
- Labels: All labels used in the called routine (FORMATs, CONTINUEs, DOs, and so on) are changed so that there is no conflict with labels in the caller.
- Actual vs. dummy parameters: Dummy parameters are replaced with their corresponding actual arguments. In the case of expressions passed as actual arguments, CF77 creates a temporary variable to hold the expression and uses the temporary in each of the places the dummy argument appeared in the called routine.
- Returns: RETURN statements are changed into branches to a new label in the caller that represents the end of the called routine. If it is an alternate RETURN statement, the branch corresponding to that RETURN is directly to the specified label.
- Function returned value: References to the function name as a variable in an expanded function are replaced with another name. (Calls to the original function may still exist unexpanded.)

All in-line expansion messages appear as warnings. CF77 does not expand any routine that has caused the generation of any of the messages except for the ROUTINE EXPANDED message.

CF77 identifies routines that have been expanded and tells you why a routine was not expanded. A summary of the routines and all the locations where a routine was or was not expanded is displayed.

User messages
B.3.6.1

The following user messages are issued in regard to in-line expansion:

- `EXPANSION ROUTINE NOT FOUND`

The expansion routine cannot be located, as specified by the `SEARCH` directive or by the default location.

- `SUBROUTINE/FUNCTION NOT FOUND IN INPUT FILE`

The input file has been defined as the location for expansion routines/functions, either by default or through a switch. The routine/function is not found in the input file.

- `EXPANSION ROUTINE IS TOO BIG FOR AUTOMATIC EXPANSION`

The routine has more noncomment lines than the allowable (the default is 50; you can change the default number of lines); use explicit expansion mode to expand. To change the default, use `cf77 -Wd"-M"`.

- `EMBEDDED CALL STATEMENT ENCOUNTERED WHILE EXPANDING`

The expansion routine references another subroutine; use explicit expansion mode to expand (for example, the `NEXPAND` directive.)

- `SYNTAX ERROR ENCOUNTERED IN EXPANSION ROUTINE`

The expansion routine has a syntax error; therefore, it is not expanded.

- `ARGUMENT MISMATCH BETWEEN CALL AND EXPANSION ROUTINE`

The arguments specified in the calling sequence of a subroutine or in a function reference do not match with arguments of the expansion routine/function. For example, this could result from a mismatch of data types, or from arrays with differing dimensions.

- EXCEEDED MAXIMUM NUMBER OF EXPANDED ROUTINES

The maximum number of different routines/functions CF77 will expand has been exceeded. Currently this number is 600. This message can also be issued if you exceed the maximum number of times a given routine/function can be in-lined; this is currently limited to 100 times.

- COMMON BLOCK MISMATCH BETWEEN CALLING AND EXPANSION ROUTINE

Common blocks found in calling and expansion routines or functions do not agree, or a COMMON variable in the expansion routine is used as a local variable in the calling routine. The common block in question is listed with this message.

- FUNCTION IN EXPANDED ROUTINE CONFLICTS WITH NON-FUNCTION

A function found in the expanded routine is in conflict with a nonfunction element in the calling routine. The function name in question is listed with this message.

- SAVE STMT. INHIBITS EXPANSION, CHANGE TO COMMON STMT.
- ENTRY STATEMENTS INHIBIT EXPANSION
- ACTUAL ARG. DIMENSIONS ARE LESS THAN DUMMY ARG. DIMENSIONS
- DATA TYPES OF ARGUMENTS DO NOT MATCH
- SCALAR ACTUAL ARGUMENT PASSED TO DUMMY ARRAY ARGUMENT

The formal argument (from the caller) is a scalar, but the dummy argument (in the callee) is declared as an array.

- EXPAN. INHIBITOR - DATA STMT. IMPLIES SAVING OF LOCAL VAR.
- WIDTH OF DUMMY ARRAY DOES NOT MATCH WIDTH OF ACTUAL ARRAY
- EXPANSION INHIBITED:SUBROUTINE USED AS A FUNCTION REFERENCE
- EXPANSION INHIBITED: FUNCTION USED IN A CALL STATEMENT

- EXPAN. INHIBITOR - DATA TYPES IN COMMON BLOCK DO NOT MATCH

The same common block appears in both caller and callee, but data types of some elements of the block are not the same in both routines.

- IN-LINE SAFE SWITCH INHIBITS EXPANSION WITH CHARACTER ARGS

When the "safe" inlining option is used (`cf77 -Wd"-e6"`), routines with character arguments are not in-lined.

- EXPAN. INHIBITOR - CONSTANT ARG. IS DESTINATION OF A READ
- EXPAN. INHIBITOR: CHARACTER SUBSTRING LENGTH UNDETERMINABLE
- WIDTH OF DUM. ARRAY AND/OR ACT. ARRAY CANNOT BE DETERMINED
- FUNCTIONS IN ARITHMETIC IF STATEMENTS ARE NOT EXPANDED
- POTENTIAL DISCREPANCY BETWEEN ACTUAL AND DUMMY ARRAY WIDTHS

The situation that triggers this warning is discussed on page 132.

- ROUTINE EXPANDED

This warning message is issued whenever a routine/function is expanded.

Index

Index

A

B

C

D

E

F

G

H

I

K

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Reader's Comment Form

CF77 Compiling System, Volume 3: Vectorization Guide SG–3073 5.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with Cray Research computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name _____ Address _____
Title _____ City _____
Company _____ State/Country _____
Telephone _____ Zip code _____
Today's date _____ Electronic mail address _____

Cut along this line

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

ATTN: Software Information Services
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold