84002390

CYBERNET® SERVICES

CYBER 203/205

User Guide

CONTROL DATA

REVISION RECORD

REVISION	DESCRIPTION
A	Manual released.
(7-17-81)	

Publication No.
84002390

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning thi
manual to:

CONTROL DATA CORPORATION
Publication and Graphics Divisi
Headquarters Publications Writi
P.O. Box 0 HQC02C
Minneapolis, Minnesota 55440

or use Comment Sheet in the bac
of this manual.

PREFACE

The objective of this document is to provide an easily digestable presentation of aspects of the usage of the CYBER 203 and the CYBER 205 that would be of immediate interest to the application programmer who has grown a little too big for conventional scalar computers. The language is informal, and the sometimes unavoidabl technical details are mostly presented in a simplified way, occasionally padded with a little white lie for honorable reasons. Consequently, this document is not to be regarded as a Control Data endorsed technical description.

In the current edition, references to the CYBER 205 is limited to Part III (Chapters 12-17), but that is only a reflection of the time-span during which the text was written. The material is equally valid for the CYBER 205 and changes will later be made to that effect.

The current operating system is today OS 1.4, but an upgrade to OS 1.5 will soon take place. Such an upgrade will require a significant updating of chapters 7 and 10, plus small corrections/additions to chapters 5, 6, 8, 9, and 11.

Timings of most code-sections in chapters 8-14 are made on the basis of compilation by the CYBER 200 FORTRAN compiler, Release 1.4, but should not be expected to differ significantly in a Release 1.5 environment. It should also be pointed out that the CYBER 203 and the CYBER 205 will yield identical execution times for scalar code. The vector processors, however, differ significantly between the two machines - a fact that should be apparent from the timing information given in chapters 15-17.

TABLE OF CONTENTS

PART I - BASICS

PART II - THE OPERATING SYSTEM

PART III - OPTIMIZATION

APPENDICES

PART I

BASICS

1.0 INTRODUCTION

The CYBER 203 is a computer. The operating system provides you with a compiler that can compile your FORTRAN program. A loader is also available, so that you can get your object code loaded and executed. You talk to the machine by feeding a card reader with punched cards, and by looking at the output that comes off the printer.

At first glance, the features mentioned above are exactly what you need, and may be all that you need. With the additional information that the CYBER 203 possesses vector processing capability, implying execution speeds far above that of other conventional computers, your immediate reaction might well be to just take all your boxes of cards, read them in and start making all your production runs on the CYBER 203 - with greatly enhanced performance. And then you could of course also enlarge the size of your model, because, as you may already know, the CYBER 203 works with virtual memory, which essentially removes all your core size boundaries.

Well, at this point we have to make you a little disappointed. Not that we don't think that the CYBER 203 is a great computer, because we do indeed think that it is. But there are no free lunches. To just try to make the move indicated above, without expecting to have to learn new things, is not realistic. Different ways of thinking have to be invoked when you deal with the CYBER 203. It represents a new generation of computers, and, as such, requires a new set of concepts in order to describe it - and an understanding of these in order to deal with it.

Consequently, to try to talk to the CYBER 203 the way you used to talk to other conventional computers is bound to get you into trouble. On the other hand, if you take the time and try to understand this new creature, it will most certainly reward you by cheaper monthly bills for more work on larger problems.

2.0 THE CYBER 200 COMPUTER SYSTEM CONFIGURATION

Although the CYBER 203 is the one that is going to do most of the work for you, it's a much too sacred creature to be dealt with directly. Instead your contact point will be either one of two front-ends: a CYBER 74 or a CYBER 175. Both the front-ends are, in turn, via a Link-station, connected to the CYBER 203. The Link-station is the turnpike through which all your job cards, source cards, input data and results have to pass. How the Link-station works, you don't really need to know - just the realization that it's there will be sufficient. The front-ends, however, as your prime contact points, do require some attention before we can start dealing with the CYBER 203 itself.

2.1 THE FRONT-ENDS

Two front-ends are currently in use. One of them is a CYBER 74 using the operating system NOS, while the other one is a CYBER 175, using SCOPE 3.4. The manuals pertinent to each of these two machines are listed in Appendix A, and you are urged to obtain the ones that are relevant to you. As you are probably aware, Control Data has adopted NOS as "The" operating system, and you are therefore encouraged to use the NOS rather than the SCOPE front-end. Phone numbers with which to connect with either one of the two are listed in Appendix B. Note, in particular, that from most cities you can dial a local phone number and get in touch with the NOS front-end. Clearly that is an additional point in favor of NOS.

When you obtain an account number (charge number + username + password) for CYBER 203 processing, you should make sure that you also get validation for the front-end you plan to use, or, better, for both of them. Note also that there is a difference between batch and interactive validation - you need both. And by all means, don't forget to obtain sufficient information about connect and login procedures.

Each of the two front-ends can be treated as separate computers, independent of each other and of the CYBER 203. When you submit

2.0 THE CYBER 200 COMPUTER SYSTEM CONFIGURATION
2.1 THE FRONT-ENDS

a NOS job to the CYBER 74 for instance, it will indeed behave just like an ordinary conventional machine. However, we will in general think of the front-end as something that enables us to process on the CYBER 203. With respect to that, the purpose of any one of the two front-ends is to allow you to do the following things:

1) Create and keep permanent files on the front-end itself.

2) Modify these files, primarily by means of UPDATE, XEDIT (CYBER 74 NOS), or the INTERCOM 5 Editor (CYBER 175 SCOPE).

3) Implement magnetic tape storage.

4) Submit jobs to the CYBER 203.

5) Receive output from the CYBER 203.

In this document we will only discuss the two last items, since the others are sufficiently treated in the corresponding reference manuals.

2.2 THE CYBER 203

The CYBER 203 is going to be your actual number cruncher. It has a FORTRAN compiler that closely adheres to the standards of FORTRAN Extended, version 5. That will mean, in general, that certain syntactical changes have to be performed. To help you with most (but not all) of the necessary conversions, the "FTN4-5 Conversion Aid" program is available on the NOS, but not on the SCOPE, front-end. Information on how to use that tool can be found in Chapter 3 and Appendix E.

The storage space available on the CYBER 203 is primarily intended for job duration files. With the possible exception of some binary files, permanent files should be stored on the front-end.

The two front-ends are both 60-bit machines with 10 characters per word, while the CYBER 203 is a 64-bit machine with 8 characters per word. When you transfer coded files (such as INPUT or OUTPUT) through the LINK station, an automatic, reversible, conversion between the two formats takes place.

2.0 THE CYBER 200 COMPUTER SYSTEM CONFIGURATION
2.2 THE CYBER 203

Binary files, however, have to be used on the machine where they were created - there is not a one-to-one correspondence between 60-bit and 64-bit floating-point formats. You may still transfer a binary file through the LINK (without conversion), but it will not be readable until it's transferred back to the machine that created it.

There is currently no magnetic tape system on the CYBER 203, but that creates no problem since you can always transfer a given file over to one of the front-ends and store it on tape there.

The operating system currently used is O.S. 1.4, which of course will be upgraded as new operating system releases become available. The reference manual (Appendix A) comes in two volumes, but only volume 1 will be of interest to you. The second volume is geared more towards systems people, and contains little useful information for the average FORTRAN programmer.

3.Ø REQUIRED CHANGES TO YOUR FORTRAN SOURCE CODE

CYBER 2ØØ FORTRAN is, if we disregard the parts that have to do with vectors, very close to FTN-5 (FORTRAN, version 5). Consequently, since your code probably is written to compile by an FTN-4 (FORTRAN Extended, version 4) compiler, some changes have to be made before you can run. In the remainder of this chapter you will find lists of various types of differences between FTN-4 and CYBER 2ØØ FORTRAN, and short descriptions on how to perform the necessary changes. The items that need attention are, somewhat arbitrarily, divided into 6 different types. If you compile your code directly, without any conversion, the FORTRAN listing will contain error messages relating to most of the items of type 1-3 that need to be changed, and that is therefore often a good practical approach. The types 4-6 are of different nature, though, and the compiler will not help you with them. Note that aborts or erroneous answers may result if you ignore these items.

On the NOS front-end (CYBER 74) there is a program installed that can help you with the conversion. Its primary purpose is to convert from FTN-4 to FTN-5, but it can also do a lot of good in converting from FTN-4 to CYBER 2ØØ FORTRAN. The items in the lists below that will be appropriately converted by this "FTN4-5 conversion aid" are marked with a double asterisk (**), while those that are only partly (or somewhat incorrectly) changed are marked with a single asterisk (*). A detailed description on how to use the program is given in Appendix E.

3.Ø REQUIRED CHANGES TO YOUR FORTRAN SOURCE CODE
3.1 TYPE 1 - PURE SYNTAX

3.1 TYPE 1 - PURE SYNTAX

** a) Remove the word TYPE preceding the words INTEGER, REAL etc., in type declaration statements.

** b) Expand the abbreviation DOUBLE to DOUBLE PRECISION.

** c) Change numeric COMMON block names to alphanumeric names starting with a letter.

** d) Change the form (vlist=dlist) in DATA statements to standard form.

e) Change the form rf*(d1,d2,...,dn) in DATA statements to standard form.

f) Make sure that the number of items in the variable lists in DATA statements exactly match the number of items in the corresponding data lists (value lists).

** g) Change the abbreviations .T., .F., .N., .A., .O., to .TRUE., .FALSE., .NOT., .AND., .OR..

h) Split multiple assignment statements; e.g. A=B=C has to be changed to two separate statements.

** i) Only one statement per line is allowed; e.g. A=B$C=D must be split into two separate statements.

** j) Change $ in column 1 to C for comments.

k) Change * in column 1 to C for comments.

** l) No compiler directives are recognized. Delete cards with C$ and C/ in columns 1-2 (not necessary).

m) Replace RETURN statements in main programs with STOP statements.

** n) Remove continuations of END statements.

o) Change $ to & in INPUT lists for NAMELIST.

3.2 TYPE 2 - MISSING OR DIFFERENT LANGUAGE COMPONENTS

a) Remove LEVEL statements. The CYBER 2Ø3 has no extended core storage (ECS) - everything is just one big virtual space.

b) Remove all OVERLAY statements and make other appropriate changes - the OVERLAY concept does not exist in CYBER 2ØØ FORTRAN.

** c) Change two-branch IF statements to one or three-branch IF statements.

d) Replace integer expressions for the index in computed GOTO statements by integer variables.

e) Complex exponents are not permitted. Change to real.

** f) Complex operands in relational expressions are not permitted. Change to real.

** g) Complex variables in arithmetic IF statements are not permitted. Change to real.

h) Replace octal constants (e.g. 1ØB) by decimal or hexadecimal constants. A hexadecimal constant may only be specified in DATA statements, and there only in the form X'123ABC'.

3.3 TYPE 3 - I/O RELATED ITEMS

a) Octal (rOw) FORMAT conversions are illegal.

b) FORMAT conversions specifying exponent length (srEw.dEe) or minimum number of digits (rIw.z) are illegal.

** c) Fw, Ew, Gw and Dw conversions in FORMAT statements are illegal, and must be replaced by Fw.Ø, Ew.Ø, Gw.Ø and Dw.Ø respectively.

* d) Equal signs and variable type (V) conversions in FORMAT statements are not permitted.

** e) nX and Tn in FORMAT statements may not be specified with n=Ø. Delete ØX and change TØ to T1.

f) Replace ≠string≠ for Hollerith constants with 'string'. *string* is acceptable.

** g) Change STOP"MESSAGE" to STOP'MESSAGE'.

h) Remove Hollerith constants in output lists, e.g. PRINT 1,6HHELLO.

i) Change list directed I/O statements to the type which reference FORMAT statements.

j) The unit identifier in I/O statements must be changed from display code in L-format to integer variables or integer constants.

k) The Hollerith L-format does not exist. Use R, H or A.

** l) Change PRINT and PUNCH with unit designators to WRITE.

** m) Change WRITE without unit designator to PRINT.

3.4 TYPE 4 - RUN TIME DANGERS

a) The file specifications on the program card must always be in the form "TAPEn=lfn", where n is a logical unit number (Ø-99) and lfn is a local file name. As an example, let's assume that you have a local file called TAPE7, and that you want to reference it with "READ(7)" or "WRITE(7)". Then you must declare "TAPE7=TAPE7" on your program card; an attempt to get by with only "TAPE7" will cause an abort at run time. The files INPUT, OUTPUT and PUNCH constitute special cases, and may appear without the part "TAPEn=". However, no default unit numbers are assigned to these special files.

* b) The number of parameters in a CALL statement must exactly match the number of parameters in the referenced entry point - secondary entry points do not automatically assume the parameter list defined in the SUBROUTINE statement.

Example:
```
SUBROUTINE X(A,B,C)
A=(B+C)**2
RETURN
ENTRY XX(A,B)
A = B**2
RETURN
END
```

This subroutine can be called with CALL X(R,S,T) or CALL XX(P,Q) but not with CALL XX(P,Q,R).

* c) Multiple RETURN's are handled differently, and must be changed as in the following examples [FORTRAN manual, page 5-4]:

FTN-4
```
CALL X(A,B), RETURNS(5,1Ø)
.
SUBROUTINE X(A,B), RETURNS(M,N)
.
RETURN M
.
RETURN N
.
RETURN
END
```

CYBER 2ØØ
```
CALL X(A,B,&5,&1Ø)
.
SUBROUTINE X(A,B,*,*)
.
RETURN 1
.
RETURN 2
.
RETURN
END
```

** d) Double exponentiation, e.g. A**B**C, is evaluated as A**(B**C) on the CYBER 2Ø3, rather than as (A**B)**C. Insert parentheses appropriately.

* e) If, in a computed GOTO statement, the index is out of range, a transfer to the next executable statement will occur, rather than an abort. Insert your own error code.

3.5 TYPE 5 - MISSING AND DIFFERENT FUNCTIONS/SUBROUTINES

a) In the statements "X=DATE(Y)", "X=SECOND(Y)" and "X=TIME(Y), only X receives the value requested - Y is a dummy parameter.

* b) The EOF-function is missing. Instead there is an END-parameter in the READ statement. (FORTRAN, page 8.1-2)

c) The LENGTH-function returns the number of 8-bit bytes (characters) rather than the number of words (FORTRAN, appendix G-2).

d) The LOCF-function is not available, but can easily be simulated using descriptors.

e) The mass storage I/O routines OPENMS, STINDX and others are missing, due to the nature of the operating system. Simulations of these routines are available on pool FTNUTIL for compatibility reasons only. See Appendix F.

f) Debug routines like DUMP, STRACE and SYSTEM are not available.

g) CYBER Record Manager routines like FILExx, STOREF and CLOSEM are not available.

h) Other routines not included in the system are:

CHEKPTX	CONNEC	DISCON	DISPLA	EXIT
IOCHEC	JDATE	LABEL	MOVLEV	OVERLAY
READEC	RECOVR	REMARK	SLITE	SLITET
SMMERGE	SMSEQ	SMSORT	SSWTCH	WRITEC

3.6 TYPE 6 - MACHINE DEPENDENT ITEMS

When software discrepancies, such as different compilers or system libraries, are not sufficient to explain the fact that a particular code produces different results when executed on two different computers, then the code could be categorized as a machine dependent code. Very often such a code manipulates data in a way that exploits the programmer's knowledge about the word size, the number representation or the data representation, and is in general quite acceptable to the compiler. It therefore becomes important to learn how to identify suspicious areas within your program - before you have wasted money on garbage runs. Look out for the following types of things:

- Masking expressions
- Shift operations
- Logical operators used with intentions other than creating logical variables
- Equivalenced real and integer variables
- Initialization of real and integer variables, when the difference in data types is ignored
- Plus and minus zero
- Hollerith variables

When you have read the next chapter, you should hopefully be capable of changing machine dependent code to a form that will produce correct results on the CYBER 203.

4.Ø DATA REPRESENTATION

4.1 INTRODUCTION

The CYBER 2Ø3 is, as previously mentioned, a 64-bit machine. With a word length of 64 bits, it is no longer convenient to use octal (base 8) notation. The hexadecimal (base 16) number system serves our purposes much better, since a hex digit requires 4 bits, and we thus can fit an even 16 digits into a full word. The 16 hex digits are Ø123456789ABCDEF.

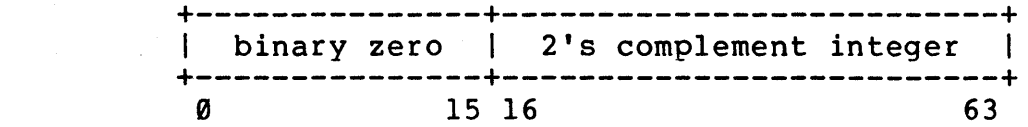
The bits are numbered from left to right, which is a consequence of the bit addressing capability - a great asset in vector processing. The leftmost bit is bit number Ø and the rightmost one is number 63.

The full word is subdivided by an imaginary line into two parts: the leftmost 16 bits (Ø-15), often referred to as the length field or exponent field, and the rightmost 48 bits (16-33), usually called the address field or the coefficient field.

The number representation used is 2's complement, which among other things has the nice feature that there is no difference between plus and minus zero. An integer consisting of n digits would in 2's complement be negated by subtracting it from 2**n, while in 1's complement we would have subtracted it from (2**n)-1. From that we can derive a quick and easy way of negating numbers in 2's complement: Take 1's complement (subtract from all 1's) and add one to get the result. Examples of how to use the rule will appear later in this chapter.

4.2 THE INTEGER FORMAT

Integer data occupies one word of storage in the following format:



The following table shows the range in hex notation [±∞ symbolize the largest and smallest integers representable in this format]:

```
0000 7FFFFFFFFFFF      + ∞
.
.
0000 000000000002      +2
0000 000000000001      +1
0000 000000000000       0
0000 FFFFFFFFFFFF      -1
0000 FFFFFFFFFFFE      -2
.
.
0000 800000000000      - ∞
```

Note that all positive numbers have bit 16 cleared (=0), while it is set for negative numbers - we can therefore think of that bit as a sign bit. Note also that the leftmost 16 bits are always identically zero.

As an exercise in 2's complement negation, let us find the representation of the integer -3:

1) Create integer +3: 0000 000000000003

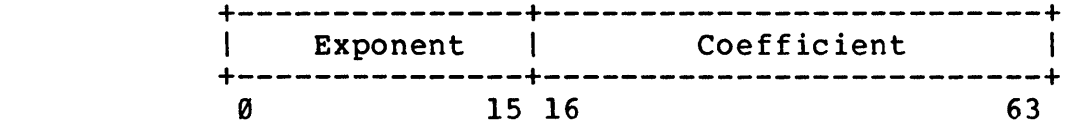
2) Take 1's complement: 0000 FFFFFFFFFFFC

3) Add 1 to get -3: 0000 FFFFFFFFFFFD

Now you may try to apply the same rules on the result to get back the +3 we started with.

4.3 THE FLOATING-POINT FORMAT

Real (floating-point) data occupies one word of storage in the following format:



Both the exponent and the coefficient are integers in 2's complement representation. The dividing line is as usual between bits 15 and 16, so that their lengths are 16 and 48 bits respectively. An imaginary decimal-point is placed after bit 63, the rightmost bit. If bits 16 and 17 are different, the number is said to be normalized. There is no bias to the exponent, i.e., the value of a floating-point number with exponent E and coefficient C is exactly C*(2**E).

To become familiar with the floating-point format, and also to get an exercise in hex arithmetic, let us construct the two numbers +1.0 and -1.0.

1) Create the integers +1 and -1:

   0000 000000000001 0000 FFFFFFFFFFFF

2) Normalize these numbers. This is accomplished by shifting the coefficients left the number of bits (NB) required to make bit 16 different from bit 17. While shifting left, zeroes are coming in from the right, and we will end up with the following coefficients:

   400000000000 800000000000

   Note that a hex 4 has the bit pattern 0100, while hex 8 looks like 1000. Hence, in each case, as required, the two leftmost bits are unequal.

3) Since the shift operations made the coefficients larger, we have to compensate by entering negative exponents, i.e. subtract the bit counts NB from the original exponents (0 in these cases). If you counted properly you should have obtained an NB of 46 for the +1 and 47 for the -1 operation. So our next problem is to

construct -46 and -47 as 16 bit integers in 2's complement representation:

a) Construct positive 16 bit integers 46 and 47:

ØØ2E ØØ2F

b) Take 1's complement to obtain:

FFDl FFDØ

c) Add 1 to get -46 and -47:

FFD2 FFDl

4) Now we are ready to form the results by merging the exponents with the coefficients.

+1.Ø = FFD2 4ØØØØØØØØØØØ -1.Ø = FFDl 8ØØØØØØØØØØØ

The permitted range of the exponent for a nonzero floating-point number is hex 9ØØØ to hex 6FFF. The zero itself is discussed in the next paragraph.

4.4 ZEROES AND INDEFINITES

The exponent range given in the last section does not cover all possible 16-bit integers. This is to allow for two special numbers: the floating-point zero and the indefinite. The definitions are as follows:

1) Any number whose leftmost hex digit is 7, is interpreted as an indefinite.

2) Any number whose leftmost hex digit is an 8, is a valid floating point zero.

That makes up for the unused portion of the exponent range. In hex notation we write the numbers as:

7XXX XXXXXXXXXXXX = indefinite
8XXX XXXXXXXXXXXX = Ø.Ø

An integer zero has the following format:

```
    0000 000000000000
```

Thus, although the 2's complement did rid us of the negative zero, we still have two different types of zeroes to play around with.

Without going too much into details, the reason why the integer zero doesn't qualify as a floating-point zero is the following: To add two floating-point numbers A and B together, their exponents must first be adjusted so that they are equal, whereafter their (adjusted) coefficients can be added and the result normalized. If B is an integer zero, i.e. has a zero exponent, and A has a negative exponent, then A is the one to be adjusted. Since an increase of the exponent corresponds to a right shift of the coefficient, that will result in a discarding of A's rightmost bits, and A+0 will in general not equal A. So the zero must have a smaller exponent than any other number, in order to guarantee that the zero is the one that gets right-shifted.

4.5 CONSEQUENCES OF THE DUAL ZERO REPRESENTATION

On most machines, there is no difference between an integer zero and a floating-point zero. But, as described in the previous section, here the difference is significant, and attention must be paid to that fact. Consider for instance the following code sequence:

```
      DIMENSION A(10),IA(10)
      EQUIVALENCE (A,IA)
      DO 10 K=1,10
   10 IA(K)=0
```

As long as you use the array name IA, and perform integer arithmetic, you are in good shape. But, since you have the EQUIVALENCE statement, chances are that you at some point will want to do floating-point arithmetic using the array name A. If you then rely on the array being zeroed, you will not get the results you expect - because A is not zeroed. If you want A to be zeroed, you have to fill it with floating-point zeroes.

To zero out a common block without paying attention to data types

is also likely to get you into trouble:

```
      COMMON /BLK/ IQ(10),N,A(10),B
      DO 10 K=1,22
   10 IQ(K) = 0
```

Like in the previous example, this will not put valid zeroes in array A and variable B.

A third type of potential error source is the data initialization performed by the system. Most (but not all) of your simple variables and arrays are zeroed out either at compile time or at load time. However, this is accomplished by clearing all the bits - which of course is worthless for floating-point data. In essence, you have to take care of all data initialization yourself, using DATA statements or executable assignment statements. Note that a DATA statement cannot be used to assign values to variables in blank common.

Please realize that you have no right to expect the system to perform any kind of data initialization that is not explicitly requested by DATA statements. It is entirely possible - and legal with respect to the FORTRAN language - that some day also the partial initialization with integer zeroes will disappear. So don't rely on what presently is standard procedure - neither on the CYBER 203 nor on any other machine.

4.6 THE HOLLERITH FORMAT

Hollerith data is stored as one character per byte, each byte being 8 bits long. That implies 8 characters per word, rather than the 10 on the front-end.

When GETPF/SAVEPF (see next chapter) are used to transfer display code files back and forth between the CYBER 203 and the front-end, an automatic, reversible, conversion takes place - unless CM=BI is specified. So the major part of the problems associated with the different byte size is taken care of by the system. However, in your FORTRAN program, you have to change all appearances of 10H... to 8H..., and maybe also A10 to A8. Instead of the changes indicated above, it may prove convenient to utilize the data type CHARACTER, which actually can handle

byte strings of both 1Ø characters and more.

4.7 HOLLERITH VARIABLES IN IF-TESTS

Consider the following code sequence:

```
      READ 1,NAME1,NAME2
    1 FORMAT (2A5)
      IF (NAME1.EQ.NAME2) GO TO 1Ø
```

If the data card has the characters HOUSEMOUSE in columns 1-1Ø, then the variables NAME1 and NAME2 will now contain [b stands for blank]:

```
+----+--------+          +----+--------+
| HO | USEbbb |    and   | MO | USEbbb |
+----+--------+          +----+--------+
```

Since the IF-test compares two integers, the content of the two leftmost bytes is of course irrelevant, and control will be transferred to statement number 1Ø - contrary to what might have been expected.

To ensure accurate treatment of Hollerith data, the data type CHARACTER should be used. Please consult the FORTRAN manual about usage.

As a temporary fix, all Hollerith variables could be declared REAL. This will force full-word compares, but may introduce other problems if and when the present ASCII character subset (64) is extended.

Yet another temporary remedy is to include the letter K in the string of options on the FORTRAN card. That will force all integer .EQ. and .NE. to 64-bit compares, but will not affect other relationals (like .GE., .LE. etc.). While the selection of the K-option thus will cure the Hollerith compares, it will also affect IF-tests containing "true" integer operands, and a small price increase may therefore show up as a side effect.

5.Ø SOME BASIC CONTROL CARDS

Before you actually submit your first job to the CYBER 2Ø3, you need to know something about the function of a few control statements. The STORE, TOC2ØØ, and TOAS cards are documented in the NOS Front-End Users Guide, the SCOPE version of the STORE nowhere accessibly, and the others in the O.S. 1.4 Reference Manual, Volume 1. Note that TOC2ØØ is a NOS control statement, while all the others can appear in CYBER 2Ø3 job streams only.

5.1 STORE

This card should be the first card in a CYBER 2Ø3 job that originates on the NOS front-end. The parameters must all appear in the proper columns, without separators. The format is as follows:

```
STORE uuuuuuaaaaaaaajjjjjjjj     x
```

Columns	Contents	Description
1- 5	STORE	Statement name
6	blank	
7-12	uuuuuu	This number is the CYBER 2Ø3 form of your CDC-supplied, batch-validated NOS front-end username. It's obtained by adding a zero in the beginning and removing the two trailing letters: 12345AA becomes Ø12345.
13-2Ø	aaaaaaaa	This number, which may appear anywhere in the field, should be your CDC-supplied charge (account) number.
21-28	jjjjjjjj	This character string, which may appear anywhere in the field, will become the name of your jobfile. The first 5 characters will appear on the banner page of your output.

5.Ø SOME BASIC CONTROL CARDS
5.1 STORE

29-33	blank	
34	X	A single letter job priority indicator. A blank or a B indicates normal batch processing. S requests standby, i.e. slower turnaround and lower charge. Priority validated users (everyone except CDC employees) may specify P for faster turnaround. Currently the cost ratios are P:B:S = 15:12:1Ø.
35-78	blank	
79-8Ø	(optional)	Keypunch indicator: 26 or 29. Default is 26.

When a CYBER 2Ø3 job originates from the SCOPE front-end, the first card will look a little different from the STORE card, although almost the same information will appear. Note that the format of this card more resembles ordinary control cards, with no column counts or embedded blanks. The following form should be used:

```
jjjjjjj,STSTR.U=uuuuuu,A=aaaaaaa,xx.
```

jjjjjjj	Job identifier. Same function as on the STORE card.
STSTR	The first two characters stand for station, the last 3 make up a logical station identifier. Currently APP is used by CDC employees and STR by customers. A third identifer, OVN, that can be used by both categories, requests OVerNight processing. Note the period after STSTR.
uuuuuu	Username as described for the STORE card.
aaaaaaa	Charge number as described for the STORE card.
xx	Job priority indicator as described for the STORE card. Here the abbreviations ST, B and PR are used for standby, Batch and PRiority. Default is B.

5.2 TOC200/TOAS: INTERFACE WITH THE NOS FRONT-END

TOC200 is a control statement that can appear in a NOS batch job, or be issued interactively. It should be used to send job decks over to the CYBER 203, alone or together with other files. The following formats can be used:

TOC200. or TOC200(I,lfn1,...)

In the first case, the default file INPUT (everything after the current position of the pointer) is sent over to the CYBER 203 as a job deck. In the second case the data files lfn1, lfn2 etc. are also sent over. They will be made local on the CYBER 203 - in time to be accessed and used by the job in file INPUT. The I may be replaced by another file name if desired. Consult the manual for more details and options.

The control statement TOAS can appear in a CYBER 203 job and works in the direction opposite to TOC200. The parameters are the same.

5.3 GETPF/SAVEPF: INTERFACE WITH THE SCOPE FRONT-END

GETPF,lfn,pfn,ID=id,ST=ADA.

SAVEPF,lfn,pfn,ID=id,ST=ADA.

lfn = local file name on the CYBER 203. (Default is lfn=pfn).
pfn = permanent file name on the front-end.
id = ID under which the file is (to be) stored.
ADA = logical identifier for the front-end.

GETPF obtains a copy of the front-end permanent file, and makes that copy a local file on the CYBER 203. A file that is attached on the front-end is not accessible to GETPF, unless the ATTACH was issued with MR=1 (multiple read access) specified.

SAVEPF makes a copy of a local or attached permanent file on the CYBER 203, and cataloges the copy as a permanent file on the front-end. Unfortunately the file is saved under a system account, and will therefore disappear at the end of the day,

unless it's moved to your own account. Such a move could be accomplished by executing the following two control statements on the front-end ["ac" symbolizes the first five digits of your account number]:

```
ATTACH,lfn,pfn,ID=id.
RENAME,lfn,pfn2,ID=id2,AC=ac.
```

5.4 FORTRAN

```
FORTRAN,I=lfn,B=bfn.
```

lfn = local file containing the source. (Default is INPUT)

bfn = Local file to receive the object code. (Default is BINARY)

5.5 LOAD

```
LOAD,bfn-list,CN=cfn,OU=dfn.
```

bfn-list = list of up to 1Ø object files. (Default is BINARY)

cfn = controllee file name, i.e. the file to receive the executable loader output. (Default is GO)

dfn = local file to receive the load map. (Default is OUTPUT). To suppress the map, make dfn a dummy name.

5.6 GO

```
GO.     or    GO(TAPEn=lfn)
```

This is the execution statement - the name of the controllee file followed by a period will do the job. Optionally you may make one or several file name replacements of the form shown, e.g. TAPE6=OUTPUT.

6.Ø SAMPLE JOB DECKS

6.1 STRAIGHTFORWARD COMPILE AND RUN

The simplest, but maybe not the most convenient, way to submit a job from the NOS front-end to the CYBER 2Ø3 is to prepare a batch deck as follows:

NOS:

```
JOB.
USER,12345AA,password.
TOC2ØØ.
7/8/9
STORE Ø12345 88888AB MYJOB       P
C2Ø3,T5Ø.
FORTRAN.
LOAD.
GO.
7/8/9
  .
  your program
  .
7/8/9
  .
  optional data cards
  .
6/7/8/9
```

The first two cards are just the normal NOS job and account cards. The third card requests that everything after the current position of the pointer in the INPUT file should be sent over to the CYBER 2Ø3. In this case that means everything that appears after the first 7/8/9 card.

The CYBER 2Ø3 job itself always (if originated from the NOS front-end) starts with a STORE card, which essentially contains accounting information. The next card is the actual CYBER 2Ø3 job card, with a decimal time limit. The FORTRAN, LOAD, and GO statements are used with default values for all parameters. Refer to Chapter 5 for a discussion of the STORE card.

6.Ø SAMPLE JOB DECKS

6.1 STRAIGHTFORWARD COMPILE AND RUN

When using the SCOPE front-end, there is no need to have a SCOPE job card sequence preceding the CYBER 2Ø3 job. All that's needed is the "distorted" STORE card, so that the whole job appears as follows:

SCOPE:

```
MYJOB,STSTR.U=Ø12345,A=88888AB,PR.
C2Ø3,T5Ø.
FORTRAN.
LOAD.
GO.
7/8/9
  .
  your program
  .
7/8/9
  .
  optional data cards
  .
6/7/8/9
```

6.2 HOW TO ACCESS SOURCE FILES FROM THE FRONT-END

A more convenient way to run a CYBER 2Ø3 job is probably to use card images stored in a permanent file on the front-end. Assuming that you have a source file called MYSRC, the procedure would then be as follows:

NOS:

```
JOB.
USER,...
GET,MYSRC.
TOC2ØØ(I,MYSRC)
7/8/9
STORE Ø12345 88888AB MYJOB       P
C2Ø3,T5Ø.
FORTRAN,I=MYSRC.
LOAD.
GO.
7/8/9
  .
  optional data cards
  .
6/7/8/9
```

6.0 SAMPLE JOB DECKS
6.2 HOW TO ACCESS SOURCE FILES FROM THE FRONT-END

SCOPE:
```
MYJOB,STSTR.U=012345,A=88888AB,PR.
C203,T50.
GETPF,A,MYSRC,ID=MINE,ST=ADA.
FORTRAN,I=A.
LOAD.
GO.
7/8/9
  .
  optional data cards
  .
6/7/8/9
```

With both source and data as permanent files on the front-end, the following decks could be used [your PROGRAM card is assumed to contain TAPE5=INPUT]:

NOS:
```
JOB.
USER,...
GET,MYSRC.
GET,MYDATA.
TOC200(I,MYSRC,MYDATA)
7/8/9
STORE 012345 88888AB MYJOB        P
C203,T50.
FORTRAN,I=MYSRC.
LOAD.
GO(TAPE5=MYDATA)
6/7/8/9
```

SCOPE:
```
MYJOB,STSTR.U=012345,A=88888AB,PR.
C203,T50.
GETPF,MYSRC,ID=MINE,ST=ADA.
GETPF,MYDATA,ID=MINE,ST=ADA.
FORTRAN,I=MYSRC.
LOAD.
GO(TAPE5=MYDATA)
6/7/8/9
```

Note that TOC200, as well as GETPF, creates local files on the CYBER 203, contrary to what is stated in one of the reference manuals. The files will thus disappear at the end of the job, unless they are made permanent with DEFINE.

6.3 A FRONT-END AND A CYBER 2Ø3 JOB IN ONE

Often times you want to edit your source and make a CYBER 2Ø3 run without having to submit two separate jobs. If you use the NOS front-end, that only requires a slight modification of the jobs in the previous section:

NOS:

```
JOB.
USER,...
GET,OLDPL=MYPL.
UPDATE,F,N.
REPLACE,NEWPL=MYPL.
GET,MYDATA.
TOC2ØØ(I,COMPILE,MYDATA)
7/8/9
  .
  update directives
  .
7/8/9
STORE Ø12345 88888AB MYJOB       P
C2Ø3,T5Ø.
FORTRAN,I=COMPILE.
LOAD.
GO(TAPE5=MYDATA)
6/7/8/9
```

First the old program library MYPL is attached under the local file name OLDPL. The OLDPL is modified by UPDATE, using the directives in the first record after the NOS control cards. The new program library NEWPL is saved as a new version of MYPL (the old version is erased by REPLACE), and then the data file MYDATA is attached. TOC2ØØ takes the rest of the INPUT file (starting with STORE) plus the UPDATE generated local file COMPILE and the data file, and sends them over to the CYBER 2Ø3.

6.0 SAMPLE JOB DECKS
6.3 A FRONT-END AND A CYBER 203 JOB IN ONE

To accomplish the same thing from the SCOPE front-end, the ROUTE statement must be used to place the job file in the input queue:

SCOPE:

```
JOB.
USER,...
ATTACH,OLDPL,MYPL,ID=MINE.
UPDATE,F,N.
CATALOG,NEWPL,MYPL,ID=MINE.
PURGE,OLDPL.
CATALOG,COMPILE,ID=MINE.
COPYBF,INPUT,JOBDK.
ROUTE,JOBDK,DC=IN.
7/8/9
  .
  update directives
  .
7/8/9
MYJOB,STSTR.U=012345,A=88888AB,PR.
C203,T50.
GETPF,COMPILE,ID=MINE,ST=ADA.
GETPF,MYDATA,ID=MINE,ST=ADA.
FORTRAN,I=COMPILE.
LOAD.
G0(TAPE5=MYDATA)
PURGE,COMPILE,ID=MINE,ST=ADA.
6/7/8/9
```

PART II

THE OPERATING SYSTEM

7.Ø THE CONCEPT OF FILES

7.1 INTRODUCTION

All CYBER 2Ø3 files are stored on disk - magnetic tape storage is currently not implemented. From the point of view of the operating system, a disk file has the following characteristics:

a) Name: 1-8 alphanumeric characters, starting with a letter.

b) Owner.

c) Length in blocks (1 block = 512 words).

d) Type: physical (P) or virtual code (C).

No logical or physical structure, such as a division in records, exists. Hence, no file positioning is possible, and commands like REWIND, COPYBR, and SKIPF etc. become meaningless.

However, when the System Record Manager (SRM) writes a file, a logical structure (recognizable only by SRM) may be imposed. From inside a FORTRAN program most file references go through SRM, and the statements REWIND, BACKSPACE and ENDFILE are thus legal and meaningful in CYBER 2ØØ FORTRAN. The FORTRAN file concept will be discussed in the last section of this chapter.

7.2 OWNERSHIP

There are three different ownership categories on the CYBER 2Ø3:

a) Public files, which can be accessed by any user. Compilers, loaders and other system utilities belong to this category.

b) Pool files, which can be accessed by users who have been

granted such permission by the owner. A pool is a collection of files under a common name, and can be created by any user. Please consult the reference manual (OS 1.4, V.1) for details, noting, in particular, the descriptions of all control statements starting with P (PCREATE, PATTACH etc.).

c) Private files, which can be accessed by the owner only - as defined by the user number of the job creating it. A private file can be either local or permanent. A permanent file can be either attached or unattached.

7.3 SEGMENTS AND EXTENSIONS

A file on the CYBER 2Ø3 can consist of up to four segments of contiguous disk space. Each of those segments must reside on the same disk drive. At creation time, the system tries to fit the whole file into one contiguous space, i.e. to use only one segment. If unable to do so, i.e., if the largest available space is smaller than the size of the file, it will split the file into two parts and create a two-segment file. More than two segments cannot be used at creation time; should the requested file length exceed the sum of the two largest available contiguous disk spaces, the system will abort and issue the error message "NO DISK SPACE AVAILABLE".

Let us now assume that you want to execute a FORTRAN program which writes to a file A. That implies that your PROGRAM statement looks something like "PROGRAM X(OUTPUT,TAPE2=A)", and that prior to turning over control to your program, the system must allocate disk space for file A. How much? Well if nothing is specified, a default value is used: local FORTRAN files are created 128 blocks long. The terms block and small page are often used interchangeably, since with the current system installation they both represent the same amount of storage, namely 512 words.

The space needed in this case (128 blocks) is rather small, and file A will most certainly be created in one segment. The fact that a file can consist of up to four segments then indicates that A may be extended three times by the addition of a new segment. The first extension will be granted when your program attempts to write beyond the initial "end of file" (the end of the 128th block). The length of the added segment depends on an installation parameter, and may vary between different computer

systems. But currently the sum of all permitted extensions will, subject to disk space availability, add up to 5Ø% of the original length of a given file. So file A above will (if necessary) be extended three times with 22, 21 and 21 blocks respectively. Had it been created in two segments, two 32 block extensions would have been available. Note that a segment can never be extended.

7.4 LOCAL AND PERMANENT FILES

Suppose now that we know in advance that the file A needs to be a lot bigger than the 192 (=128+64) blocks automatically allocated. Then we clearly must make sure that adequate space, say 1ØØØ blocks, is reserved already at creation time. That can be accomplished by creating the local file A prior to execution of the program, using the following control statement:

```
REQUEST,A/1ØØØ.
```

The length may alternatively be given as a hexadecimal number:

```
REQUEST,A/#3E8.
```

When the time comes for your program to execute, the file A will thus already exist, and need not be recreated.

All system utilities that need a write file (FORTRAN, LOAD, COPY etc.) have the capacity of creating a local file, and will do so if necessary. However, as we have seen, the default length may sometimes be too small, and in those cases we must create the file in question beforehand. The default lengths are, in general, indicated in appropriate sections of the reference manuals, and vary from task to task. For instance, the binary object file generated by the FORTRAN compiler is created with length #1Ø, while #1Ø2 blocks are allocated for the executable file written by the loader.

Normally a local file A will disappear at the end of the job that created it. To keep the file it has to be made permanent by issuing the control statement:

```
DEFINE,A.
```

Since here A is assumed to already exist, the only effect of the DEFINE control card is that the file name is entered into the permanent file catalog. Any parameters appearing after the file

name are thus irrelevant and will be ignored - the length, for instance, cannot be changed.

DEFINE can also be used to create a permanent file, in which case the parameter set is identical to the one used by REQUEST. The default lengths for file creations by both DEFINE and REQUEST are 8 blocks.

When a file has been created and made permanent, it's considered to be attached to the job until the last job card is processed. However, a permanent file that was created in a previous job cannot be used until it has been attached. One control card is enough to attach several files:

```
ATTACH,A,B,C.
```

To attach all your files, you can use:

```
ATTACH,*.
```

7.5 PHYSICAL AND VIRTUAL FILES

A file that cannot be executed is called a physical file or a data file. An executable file is called a virtual file or a controllee file. Currently the only utility (not counting COPY) that can write a controllee file is the loader, and it uses the default length #1Ø2 blocks and default name GO. Hence the sequence:

```
FORTRAN.
LOAD.
```

will provide you with an executable local file GO. The length of GO will automatically be decreased when the loader is done - the #1Ø2 blocks is only a creation size. In fact, most utilities shrink their output files when they have completed their task - that way you won't be charged for empty file space in case you decide to make the files in question permanent.

REQUEST and DEFINE will, by default, create physical files, but addition of the parameter "TYPE=C" or "T=C" after the "file/length" parameter causes the creation of an empty virtual file:

```
REQUEST,A/#100,T=C.
```

To inform the loader that a space is already reserved for the controllee file, the CN parameter must be used:

```
LOAD,CN=A.
```

The length of the controllee file is often more conveniently specified directly on the load card:

```
LOAD,CN=A/#100.
```

Note that in order to do that, A must not exist beforehand.

Most system utilities that create files allow the user to specify the lengths of those files directly on the control card in question. However, when many large files are needed for a job, it is safer to first create all files using REQUEST. The reason is that if not enough file space is available to accommodate all of your files, then you probably want to abort at the beginning of your job rather than at the end.

7.6 FILES IN A FORTRAN PROGRAM

A file that is declared on a program card (or added at execution time in the GO statement), and referenced from within that program, will be handled by the System Record Manager (SRM). Although a file cannot possess any structure on the level of the operating system, the SRM does impose an artificial structure on files that it writes. That structure for binary files simply consists of a key word in the beginning of each record, indicating the number of words (actually the number of bytes) contained in that record. In addition, a few other types of informative key words may be put in the file by the SRM. Formatted (ASCII) files have yet a different structure, but we will not discuss its details here.

Thanks to the SRM, file positioning statements in FORTRAN, i.e. REWIND and BACKSPACE, still make sense. The same is true for ENDFILE. It's important to realize, though, that the SRM structure is recognizable only by the SRM, and not by the system - you cannot backspace or rewind a file using control

statements. On the other hand, the structure is very real to the SRM, so that a file written by program A can be read by any FORTRAN program, including A. Making a file permanent will not destroy the structure.

Before we can discuss different types of I/O, i.e. different methods to transfer data between a program data area and a file, we need some basic understanding of the concept of virtual memory. Chapter 8 is therefore devoted to virtual memory, and not until after that do we deal with implicit and explicit I/O, the most important I/O choices available. For concurrent I/O methods (Q7BUFIN and Q7BUFOUT) we must refer to the FORTRAN manual, page 14.11-13.

8.0 VIRTUAL MEMORY

8.1 INTRODUCTION

A conventional computer usually features a central memory of a few hundred thousand words, plus, in some cases, a large core memory (LCM or ECS) of comparable size. In a multiprogramming environment that space is then to be shared between several executing jobs, limiting the size of the code and data areas of a single user to maybe 100K words or less.

On the CYBER 203 however, each user has access to a rather large virtual address space, the size of which is limited only by the size of the address field, i.e. the number of bits in a memory word that could be used to hold an address. The word partitioning for addressing purposes is 16/48 (the same as for data storage), and the rightmost 48 bits constitute the address field. Since all addresses are bit addresses (an individual bit in central memory can be toggled!) and each word contains 64 bits, the total size of the virtual address space is:

$$2^{42} = 4,398,046,511,104 \text{ words.}$$

Currently the system does not permit negative addresses, but, even so, the size of the space available to each user is impressive - more than 2 trillion words!

Now, 2 trillion words - that sounds too good to be true. And, of course, in a sense it is. Because what counts is not how many addresses we can create, but rather how many words of data we can store and manipulate. Our storage devices are of two types: a central memory with 1 million words (expandable to 2 million), and about a dozen disk packs, each having a capacity of about 33 million words. As will be explained in the following sections, virtual memory is, in a sense, the same as disk space, and it therefore makes perfect sense to say that the CYBER 203 has a user accessible memory capacity of several hundred million words - still an impressive number.

8.2 RELOCATABLE ADDRESSES

There are (at least) three different types of addresses: relocatable, virtual and physical. The latter type is the kind that actually describes where in central memory a particular machine instruction or data element is stored, while the other two more resemble a labeling scheme.

When the compiler gets hold of the card images constituting your program, it first divides it into modules. A module is a program unit, a subroutine or a function, and all such modules are treated as independent items by the compiler. As a given module is compiled, each machine instruction produced is assigned an address - a relocatable address. This is nothing but a count that keeps track of how much memory space will be required to hold this code when it later executes. The address itself is often referred to as the value of the location counter or program counter, and can be found in any assembly listing - usually in the leftmost column.

Since each module is treated independently, the first instruction (i.e. the first entry point) in a given module will always correspond to address zero. When all modules are later merged (at load time), those addresses must be changed, or relocated, to avoid confusion. That is, of course, the reason why we use the term relocatable addresses.

8.3 VIRTUAL ADDRESSES

The next step towards program execution is to gather all of your relevant modules (routines) together and build a controllee file. That's handled by the loader and requires, among other things, the relocation of all addresses into what's called virtual addresses. That will establish a new labeling scheme - one that allows all modules to be concatenated into one large chunk of code without address duplications.

As an example, if the code sections of the subroutines A, B and C were previously assigned the relocatable addresses 0-199, 0-299 and 0-99, they may now be assigned virtual addresses 3501-3700, 3701-4000 and 4001-4100.

8.Ø VIRTUAL MEMORY
8.3 VIRTUAL ADDRESSES

In the same manner, the loader will collect everything you need for execution - your own routines, system routines, library routines, FORTRAN error processing routines, etc. - and gradually fill up your own private virtual space. As you may already have guessed, it's of course not sufficient to just assign addresses - the stuff has to be kept somewhere as well. So what we are really talking about is the creation of a disk file, and the assignment of virtual addresses, starting with zero at the beginning of the file and proceeding sequentially as far as necessary. Since the file is going to be executable, we'll refer to it as a controllee file, using the special terminology of the CYBER 2Ø3.

In an effort to keep the concept simple, we have here only talked about addresses with respect to code, or machine instructions. Clearly something has to be done with the data areas as well, but let's leave that aside until we talk about the structure of the controllee file in more detail.

At this point it should be clear that the size of your virtual space by no means is limited by the size of the central memory of the machine, but rather by the capacity of available storage devices. We also note that the virtual address of a given instruction does not tell us where in the machine it later will be found - it only provides a means of describing it's location in the disk file where it is stored, and then only relative to the beginning of that file.

The loader-built controllee file thus constitutes the beginning of your virtual space, but not necessarily the end. In fact, when you really go into execution, a number of other files may well be concatenated in virtual space with your controllee file - thus expanding that space tenfold or more. Concatenation of two files in virtual space simply amounts to assigning proper virtual addresses - they need not be physically merged.

8.4 PHYSICAL ADDRESSES

The central processor can read the central memory but not peripheral storage devices such as disks. So before a given instruction can be executed, it has to reside in central memory, where the processor can get at it. This means that, at some point, the part of the controllee that contains that particular instruction has to be copied into physical (=central) memory. Since each user has his own virtual address space, starting at zero, there is now, in a multiprogramming environment, a possibility that the same virtual address will appear more than once in physical memory. So yet another address scheme must be introduced - physical addresses have to be assigned.

A physical address is simply an address describing a particular location in physical memory. Your program will never be concerned with physical memory locations - internally it can only reference virtual addresses. So there must not be an irreversible change from virtual to physical addresses; rather, a rule of translation from one type to the other must be established. That rule is called a mapping, and will simply be a prescription to add or subtract a constant.

Let's take an example based on the current memory configuration of the CYBER 203. The size of physical memory is 1,048,576 words, which in hex notation looks like #100000. Since all addresses are bit addresses, we multiply by 64 to get #4000000 bits, and that gives us a proper physical address range of 0 to #3FFFFFF. Suppose now that you are about to execute the instruction that has the virtual address #10000. The system finds (we will assume) that that part of the controllee has yet not been brought into memory. Hence, it locates a free space there, say physical location #410000 and onwards. As discussed in the next section, only pages (a small page = 512 words = #8000 bits) can be moved, so the section #10000 - #17FFF of virtual space will now be copied from the controllee into physical memory locations #410000 - #417FFF. The mapping will in this case consist of the following rule:

physical address = virtual address + #400000

The system can now, using this rule, translate any virtual address in that particular page to the correspoding physical address. Note that the subsequent virtual page #18000 - #1FFFF may well, when referenced, end up in physical locations #400000 -

#407FFF, i.e. the order between pages in virtual space does not at all have to be preserved in physical memory - each page is mapped independently.

8.5 THE DROP FILE

When the system decides that a particular instruction has to be moved into memory, it does not have a free choice about the number of words to move. Data transfer between disk and memory must take place in units of either small or large pages. A small page consists of 512 words (#8000 bits) and a large page of 65536 words (#400000 bits). The address of the beginning of a small (large) page is always zero modulo #8000 (#400000), i.e. it starts on a small (large) page boundary - in both virtual and physical space. That way it can be ascertained that a given address always belongs unambiguously to one page only.

As soon as it is established that the page you need is not in memory, you will incur a page fault. That implies that you will stop processing, and a system request goes out for that page. In general there is no free space in memory, and a decision has to be made about which space you are going to steal. For such decisions, the system uses an algorithm called the LRU (Least Recently Used) algorithm, and the page that has been inactive (unreferenced) for the longest time will be the one that has to go to give you space.

Let's now assume that the page that has to go was one of your own pages, and that it contains some code that you have used once, and will use again later. The first question that arrises is: Has the page been modified in memory or not? If the answer is no, then we know that a replica of it is sitting out in the controllee file, and that the area of memory which it currently occupies can safely be overwritten. However, if it has been modified, then its new image has to be saved somewhere for later retrieval. The disk area to which the system writes modified pages is called the drop file. It is a job duration file, created just before your job goes into execution. If you, for any reason, rolled out completely during execution, the drop file together with the controllee file will hold all current page images. A restart is thus a comparatively simple affair.

Note that when a page is dropped, the virtual/physical mapping is also erased, so that the next time it's needed it may be read in

to a totally different area of memory. Hence, although the virtual address of a given instruction never will change, the physical address may very well do so.

8.6 THE CONTROLLEE FILE

As already discussed, the loader builds an executable file, which we will call the controllee file. With the exception of the first small page (the minus page), each location in that file will be assigned a virtual bit address. In order of increasing virtual addresses, the controllee will be made up of the following constituents:

Minus Page	(Not part of the virtual space)
Virtual Page Zero	(Virtual address Ø-#8ØØØ)
Relocated Code	
Data Bases	
Labeled Commons	
Error Processing Information	

The first small page, called the Minus Page, contains information that the system needs in order to execute the file. It takes up space in your disk file, but is actually not part of your virtual space.

The next small page represents your register file, i.e. your working registers, and is the page that really constitutes the beginning of your virtual space. At the level of this presentation, we can ignore that page as well.

Next comes the relocated code which, as previously discussed, is a concatenation of all your routines, plus whatever the loader decided you will need from SYSLIB or other specified libraries.

The data base section is made up of all the data bases belonging to the different routines. The data base of, say, subroutine SUBA, consists of an area in virtual space that is large enough to accommodate all local variables of SUBA. That amounts to those arrays that are neither in common nor are passed into SUBA as parameters, as well as simple variables with the same characteristics. If SUBA contains DATA statements for some of these variables, the loader will satisfy those and write the

requested values directly into the data base part of the controllee file.

Directly after the data bases we find the labeled commons. They will also be initialized by the loader, if there are data statements requesting that.

The last group of items in the controllee file consists of FORTRAN error processing information, and is hardly anything that concerns us.

To summarize, the controllee file consists essentially of code, data bases and labeled commons, and its size is thus determined by the size of those items.

When the loader is done, you may well choose to make the controllee file permanent, using DEFINE, thereby cutting down on overhead in subsequent runs. Since before each such run you will expect to find the same initial values in data bases and labeled common blocks, the controllee file is, for all practical purposes, write-protected during execution - it will never be altered.

Note that blank common is not accounted for above, i.e. it is not part of the controllee file. One implication of that is that the size of the controllee can be kept reasonably small even for codes with large arrays, provided these arrays are placed in blank rather than labeled common. Since initialization can only take place for storage space represented in the controllee, we also conclude that data statements are meaningless when they refer to blank common - in accordance with standard FORTRAN.

8.7 BLANK COMMON

Blank common is the first extension of your virtual space beyond the limits of the controllee. Before you are ready to execute, i.e. before you are in a position to reference any data element in blank common, there must be a virtual address associated with that element. So the full blank common block, as dimensioned in your program, must be assigned an area of virtual space. However, at this point (i.e. prior to any reference), that assignment is purely fictitious, and only amounts to bumping the

pointer that points to the last used address in your virtual space. This is so since there are no initial values associated with blank common, and therefore the first meaningful reference must be a WRITE - why should the system bother to reserve a particular disk area, when the content of it would be immaterial anyway?

Of course, when your program finally does write into blank common, then some actual space (physical memory as well as disk) has to be reserved - at least for the part that's about to be modified. The required pages in physical memory are given to you when you need them in the form of some uninitialized space, and at some later time these pages will be written out to the drop file. Hence the drop file has to be large enough to accommodate all used blank common space, in addition to the space corresponding to data bases and labeled common blocks (we will always assume that you do not have self-modifying code).

Since the size of the drop file has to be determined at creation time, which is just prior to execution, the system has nothing but dimension statements to go by. At that time it does not know how many elements in blank common you will reference. A consequence of this is that if you try to use more space than your dimension statements indicate, then the drop file is likely to overflow at some time. Although the system will grant you a few extentions, as for any other file, that will probably not be enough. Particularly not if you (as is common on other machines) have only one array in blank common, and that array is dimensioned to one! Note that a "drop file overflow" error is a run time error, which will cause an abort.

Although it's no fun to cause an overflow of the drop file, there are worse things. By overstepping the declared boundaries of blank common, you will enter a part of virtual space that often is reserved for other purposes. As we shall see in the next section, the files are usually mapped immediately after blank common - so by walking across boundaries you may actually alter the content of a file!

We thus conclude that blank common must be dimensioned to at least as much as you plan to use. In fact, since the virtual space in this case is fictitious, you might as well make it as big as you are ever going to need! What counts, namely, is not the size of blank common, but rather how much you use - the pages will actually not exist until they are referenced. And you won't be charged for them either. There is, however, one little catch:

the system has to reserve space in the drop file - and it doesn't know how much you are going to use. So here, as well as anywhere else, use moderation. A blank common dimensioned to a few hundred thousand words will probably never hurt you, while if you go up to several millions, it might be hard for the system to find ample disk space for the drop file.

9.Ø USER CONTROLLABLE PAGE MAPPING

9.1 INTRODUCTION

A page fault will occur every time you reference a page that is currently not in memory. Since you at that time will need something that is not yet available, processing (of your code) must be suspended until the page in question is mapped and read into memory. If the page is a large page (65536 words), then about .4 - .5 seconds wall clock (=real) time will pass from the moment when the page is referenced until the data is available. Some CP (central processor) time will also be consumed - maybe a few microseconds. A small page fault takes about .1 - .2 seconds wall clock time, while the CP time is pretty much independent of the page size.

Although the numbers given above are only rough estimates, they clearly show that the difference between 1 large and 128 small page faults is huge - both with respect to wall clock and CP time. And yet a large page can hold the same amount of data as 128 small pages. It is therefore evident that a program that manipulates large amounts of data will finish it's task much faster if the data transfers between disk and memory take place in units of large pages rather than small.

Not only data, but also code has to be transferred between disk and memory. Although the size of your code may be substantial, that does not, in itself, warrant a mapping of it onto large pages. The reason is that code, in general, is processed much slower than data, implying that a large page in memory containing code would display a very low activity. And low activity means wasted space. In a virtual memory environment, the difference between letting your code occupy 512 and 65536 words of real memory may be the difference between getting your results back in 1Ø minutes and 1Ø hours! Although that comparison may be a bit extreme, it is not unrealistic. However, the more "normal" effect of wasting space is to decrease the available space for other users when you are running, and thereby affecting, in a negative way, the turnaround - even for yourself. Since system resource utilization, such as paging, also costs, and since

9.Ø USER CONTROLLABLE PAGE MAPPING
9.1 INTRODUCTION

wasting space generally means more frequent paging, your cost will probably also increase somewhat.

9.2 THE GRLP PARAMETER

The conclusion must be that data (in particular large arrays) is better off being transferred in units of large pages, while code should be moved a small page at a time. By default, the loader maps your program (the controllee plus blank common) onto small pages. However, you may override that, by specifying on the load card that certain groups of items are to be mapped onto large pages. In this context we recognize three different types of items: modules (subroutines, functions), labeled common blocks and blank common block. These may not be mixed, i.e. all items in a group must have the same type. The format of the specification is:

```
LOAD,...,GRLP=group1,GRLP=group2,...
```

where group1 and group2 each symbolizes a list of items separated by commas. For this purpose a module is identified by name only, and a labeled common block by *name. A single * means blank common. As an example, consider:

```
LOAD,...,GRLP=*BK1,*BK2,*BK3,GRLP=*BK4,GRLP=*.
```

This would cause common blocks BK1, BK2 and BK3 to be mapped as one group (with no space between them) onto large pages, while common block BK4 and blank common each would be mapped by itself. The virtual addresses are always aligned so that each group starts on a large page boundary. The order within a group will not necessarily be the order given on the load card, and the order of the groups themselves may also be changed by the loader.

Note that it's easy to waste space in memory by doing a careless grouping. In the example above, consider the situation that the four labeled common blocks each are a quarter of a large page (16384 words) long. Then the two first groups will together occupy 2 large pages. By grouping them in one group only, that could be reduced by a factor of 2! Blank common must always appear alone, though.

9.3 GRLPALL AND GRSP

When you are just trying to get your program up and running, i.e. when producing correct results on a test case is more important than performance, then the GRLPALL parameter may come in handy. The format is (note the blank field after the equal sign):

```
LOAD,...,GRLPALL= ,...
```

If GRLPALL is specified, the loader will map both the controllee and the blank common onto large pages. Note that that includes code as well as data, and therefore is not the optimal choice. The effect of GRLP is not changed by the presence of GRLPALL.

Note that it is not possible to exclude code from being mapped onto large pages when GRLPALL is used. To specify GRSP=module-list, for instance, will not have the desired effect - the group specified by module-list will be aligned to start on a small page boundary, but the mapping will still be onto large pages.

9.4 GROS AND GROL

As you remember, blank common is not part of the controllee, and the disk space requirements are thereby reduced. A labeled common block can be forced out of the controllee in much the same way, by using GROS or GROL:

```
LOAD,...,GROS=*A,GROL=*B,*C,...
```

The effect of this is to force common blocks A, B and C out of the controllee. A will be mapped starting on a small page boundary, and the group consisting of B and C will be mapped starting on a large page boundary. Note that GROL does not map B and C on large pages - it only aligns the starting address to a large page boundary. That fact cannot be changed by another specification such as GRLP or GRLPALL. Note also that when a common block is not part of the controllee, it cannot be initialized with DATA statements.

9.5 THE DYNAMIC STACK

Although it's not transparent to the user, any FORTRAN program makes use of a certain amount of scratch storage space during execution. That space is taken out of what's called the dynamic space (dynamic stack), which is a name for the part of virtual space that begins just beyond the highest virtual address assigned at the time you go into execution.

A program that does not contain any vector statements usually requires comparatively little scratch storage - maybe a few small pages. The main use in that case is for subroutine linkage, which represents a very low frequency with respect to referencing the dynamic stack pages utilized. A pure scalar code therefore will not gain from having the dynamic stack mapped onto large pages - it may actually lose.

If your code is vectorized, and, in particular, if it was compiled with the V-option (which requests automatic vectorization), then a substantial utilization of the dynamic stack may take place. To calculate a complicated vector expression, it's namely often necessary to store intermediate vector results somewhere. And although the dynamic space is reused over and over again, a program that deals with very long vectors stands a good chance to be noticably slowed down if the stack is mapped onto small pages rather than large.

When appropriate, you may request the mapping of the dynamic stack onto large pages. It's done by specifying the RLP parameter on the PROGRAM card (not LOAD card):

```
PROGRAM X(...,RLP=2,...)
```

In the example above, the first 2*65536 words of the dynamic stack will be mapped on large pages. RLP=1 may be abbreviated to RLP only. The RLP parameter may, perhaps more conveniently, be specified on the execute card:

```
GO(...,RLP=2,...)
```

Note that a heavy use of the dynamic stack may increase the drop file size requirements. The loader has namely no way of knowing how much is going to be used, and he (she?) is the one who sets the size of the drop file. So you may have to explicitly specify

that size, using the CDF parameter on the load card.

By unnecessarily specifying the RLP parameter, you may increase the apparent use of the dynamic stack from maybe 2 small pages to 1 large page, since every time you reference it, the whole large page has to be transferred (if not in memory already). It is therefore an excellent way of wasting space if you don't use your judgement. That wasted space may well degrade performance, and it may also cause a drop file overflow (at run time!), as mentioned above.

10.0 IMPLICIT AND EXPLICIT I/O

10.1 INTRODUCTION

Program initiated data transfer is always of the type memory to memory. That goes for READ's of and WRITE's to files, as well as for ordinary arithmetic assignments. So a file, or at least the relevant part of it, must be mapped and moved into physical memory whenever an I/O statement is to be executed. There are two different ways of performing that mapping, each of them demanding its particular type of I/O - implicit and explicit. Note that when we talk below about implicit and explicit files, we do not mean to imply that the files themselves are of different types, because they are not. It's just a convenient way of describing how the files in question are linked with your program.

10.2 IMPLICIT I/O

An implicit file (or rather a file on which implicit I/O can take place) will appear in your program statement as something like TAPE3=TAPE3. Since that is the normal way to declare files on the CYBER 203, you may think of implicit I/O as the default type of I/O. The first step in linking an implicit file with your program is to map it into your virtual space. Files are mapped after blank common, and will normally start immediately after it. Note that this mapping does not encompass any reallocation of file space. The files stay where they are, and all that really happens is that virtual addresses are assigned to their content.

When, in the course of execution, you reference an implicit file, then the corresponding page will be mapped and moved into physical memory, in just the same way as when a part of the controllee is referenced. At some time later on, that particular page will have to be dropped, because the space it occupies will be needed to hold some other data. If it was never modified, i.e., if the only references were via READ statements, then it

can safely be erased. A modified page, though, must be written out to disk somewhere. In this case the drop file is not involved. Instead the page is written directly to the file in question. That's where you want the new information anyway - right?

Let's now assume the following situation: You have just assembled a particular matrix A, i.e., you have computed all of its elements. Your plan is to go on with some further processing of A, such as solving the system of linear equations that it represents. But since A is quite big - it occupies 2/3 of available memory - and it took quite a lot of work to assemble it in the first place, you would like to save an image of A in a file (TAPE3) before you proceed. The following FORTRAN statement would accomplish that:

```
WRITE (3) A
```

In the transfer of the first half of A, no problems will be encountered. Starting from the beginning of TAPE3 (or from wherever the pointer currently is), a page will be read into memory, and the corresponding part of A will be transferred to it. Those steps will then be repeated until the last third of memory is filled by pages representing the part of virtual space that is associated with TAPE3. Since the memory at that point will be completely filled up - two thirds is used for A and one third for TAPE3 - another page of TAPE3 cannot be brought in without first dropping one of the others.

Clearly the least recently used pages are the ones representing the second half of A. If we label those 1, 2, 3,..., then the page that has to go is (2). When (1) has been transferred we will need (2) plus a new page of the file, which means that (3) and (4) have to be dropped to give room. And so on. The pager does an excellent job in chasing its own tail. Whenever a new page of A is about to be transferred, it has to be read into memory, since it was just written out to the drop file. Moreover, if you give it some thought, you will realize that before the transfer is completed, the very first pages of A will also have been forced out to the drop file. The content of memory will, at that point, be the last 3/4 of A and the last 3/4 of the written file pages.

Now consider the original formulation of the problem: Dump A to TAPE3, and then go on and process A. The first thing you want to

do after the write is thus to reference A, and then probably the beginning of A. But the corresponding pages are sitting out in the drop file, and before they are available in memory again we will have incurred several page faults. Those page faults will in turn probably force out more of A, so that the process becomes even further prolonged.

The conlcusion must be that data transfer to and from an implicit file can be very inefficient when the record size is large, and we do indeed need an alternative way of handling the I/O. A situation like the one described above must be considered unacceptable.

Note that several I/O statements interleaved with regular data processing may not have the effect described above, even if the total data transfer is the same. What counts, namely, is the order in which the pages are referenced, and obviously some extra processing will alter that order. Let's also emphasize that small amounts of data, or small records, can readily be transferred using implicit I/O. It's really only the big moves that can hurt us. And big means at least several large pages.

It should also be mentioned that implicit I/O will disappear totally when Operating System 1.5 is released. That will probably happen during the first half of 1981 - but until then the default is implicit I/O.

10.3 EXPLICIT I/O

As described in chapter 7 of the FORTRAN manual, there is an alternate way of declaring a file on the program card, namely:

```
TAPE3[,,4]=TAPE3
```

The brackets indicate that all I/O on this file should be of type explicit. The first three parameters within the brackets concern the storage medium used. The combination given above - the first two omitted and the third equal to 4 - specifies a disk file, which currently is our only option.

When explicit I/O is used, data transfers initiated by READ

10.0 IMPLICIT AND EXPLICIT I/O
10.3 EXPLICIT I/O

(WRITE) statements are accomplished by the gradual emptying (filling) of a buffer, which serves as an intermediate storage area. The size of the buffer is in general 3 small pages, but can be extended up to 24 small pages by specifying a fourth parameter in the brackets, e.g.:

```
TAPE3[,,4,15]=TAPE3
```

The buffer is an area of virtual space that is mapped into physical memory whenever data transfer is taking place. The buffer pages are no different from other virtual pages, i.e., if they are not referenced for a while, the system may decide that the space they occupy is needed for some other purpose. Therefore, at a given time, some buffer pages may be out in the drop file, while the remaining ones reside in physical memory.

Assuming a buffer size of NP small pages, when the first file reference is made, the first NP pages of the file will be mapped onto the buffer and read into memory. Then, as READ's or WRITE's are issued, an imaginary pointer moves from the beginning towards the end of the buffer. When the end is reached, or rather when the word following the last mapped word is referenced, the content of the buffer will be written out to the part of the file pointed to by the mapping. Immediately following that, a mapping of the NP next pages is established, whereafter the buffer is filled with the image of those.

Note that a "read from disk" or "write to disk" (which corresponds to a swapping of the buffer content) is initiated by the system, rather than by the program, and that it's triggered by the crossing of a buffer boundary, rather than a page boundary. A swap includes all buffer pages, and, thus, cannot get started until all such pages are in memory. That means that for each dropped page a page fault will be incurred, causing the relevant image to be read into memory from the drop file. It would therefore be to your advantage if the full buffer resided in memory when a swap were about to take place, and that's one of the concerns that governs the choice of buffer size.

The longer the time between references to the first and the last page in the buffer, the higher the probability that the first page has been dropped when the last is referenced. When the record size is (almost) constant, i.e. when all I/O references (READ/WRITE) to a given file request the transfer of about the same number of words, the number of page faults can therefore be

minimized by choosing a buffer size that is close to the record size.

If the I/O on a file is dominated by the transfer of very large records, then the number of page faults will automatically be small, since, on the average, several swaps of the buffer content will be associated with each READ/WRITE. For such a file the main concern becomes to minimize the number of swaps, and that is accomplished by specifying the largest possible buffer size, 24 small pages.

10.4 WHICH IS BETTER - IMPLICIT OR EXPLICIT I/O?

Let's first summarize the differences:

1) An implicit file is mapped in its entirety into virtual space, while an explicit file at any given time has only NP pages mapped, NP being the size of the buffer. Hence an explicit file will never compete for space in physical memory, while an implicit file may well do so, and thereby force more important data out to the drop file.

2) The overhead required to initiate a READ or a WRITE is almost zero for an implicit file, while it's substantial for an explicit file.

3) An I/O request for a large amount of data will, in the implicit case, probably be interrupted quite often (every 512 words) by page faults. On the other hand, explicit I/O can, in the same situation, proceed uninterrupted until the buffer needs to be swapped - which will happen considerably less often. In addition, the swap of the buffer content is done in a very efficient manner.

The conclusion must be that implicit I/O is to be preferred when you are dealing with small records, and, in particular, when the I/O statements are interleaved with other code. In other cases, i.e., when the record sizes are large or a lot of data is to be transferred in at least moderately-sized records (several small pages), explicit I/O is the way to go.

Note that the terms implicit and explicit I/O do not describe the files themselves, but rather how they are linked to your program. Hence, the same physical file can appear as explicit

10.0 IMPLICIT AND EXPLICIT I/O
10.4 WHICH IS BETTER - IMPLICIT OR EXPLICIT I/O?

(with brackets) in program A, and as implicit (without brackets) in program B - even if A and B are executed in the same job stream. It is also possible to override the program card declaration at execution time, simply by making a new declaration in the GO-statement. For instance,

```
GO(TAPE3[,,4]=TAPE3)
```

will make TAPE3 explicit, regardless of the declaration on the program card. In the same manner, files that are not even defined in the program may be introduced at execution time.

11.Ø TASKS

11.1 INTRODUCTION

Due to the fact that the operating system is task-oriented, the three files INPUT, OUTPUT and DAYFILE do not behave quite like they do on some other systems. Rather than presenting a detailed description of the task-related features, we will here only lightly touch on what's relevant for an understanding of how these files must be handled - or rather not handled. We will, thereby, consider the performance of a task equivalent to the execution of a controllee. It's also worth noting that each control statement in your job stream names a controllee - COPY, DEFINE, SWITCH, etc., are all controllee's. Hence each of your control cards requests the system to perform one particular task.

11.2 THE INPUT FILE

The first step in the execution of a task is that an "overseer" program called BATCHPRO gets hold of the controllee. BATCHPRO then checks the job environment for the existence of file INPUT. If INPUT does not exist (as a local or attached permanent file), a file with that name is created, and the next record of the job file is copied into it. The job file, which consists of your job cards followed by a number of records separated by EOR's (7/8/9 cards), can be read only by BATCHPRO. When he copies a record into INPUT, he also moves the job file pointer to the next EOR, which makes the record irretrievable. When the task has completed, the file INPUT is destroyed if, and only if, it was opened. Otherwise it is kept alive to (perhaps) be used by the next task. The user is affected as follows:

1) File INPUT cannot be read past the end of file.

2) Two programs cannot share a file with the name INPUT.

11.3 THE OUTPUT FILE

When a task has completed, and control is returned to BATCHPRO, it is determined whether the program generated a file named OUTPUT. If such was the case, the file is renamed by BATCHPRO, and tagged to indicate its origin. That has the following implications:

1) The file OUTPUT generated by a given task (e.g. your program) is not accessible to any other task. In particular, OUTPUT cannot be copied or routed.

2) The print limit, i.e., the size of file OUTPUT, is meaningful only on a task-by-task basis.

11.4 THE DAYFILE

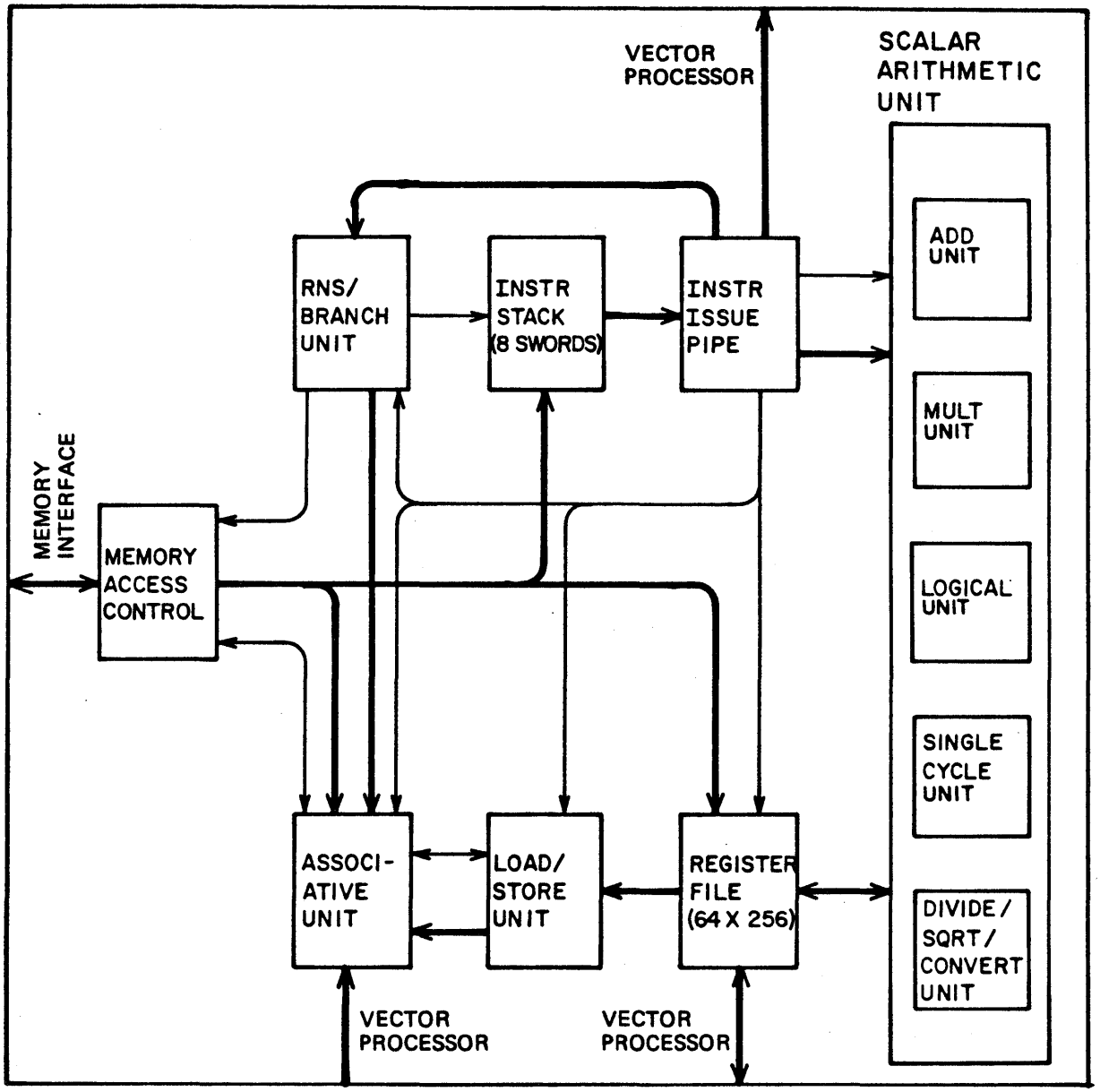
The dayfile messages created by a particular task are given to BATCHPRO. When the task has completed, those messages are written into a job-duration file called Q5DAYFLE, and the task is terminated. The dayfile is thus accumulated in one place only, in contrast to what happens with the different OUTPUT files, which just get renamed and tagged. However, only BATCHPRO has access to Q5DAYFLE - an attempt by the user to copy the contents will at best yield nothing and at worst confuse BATCHPRO by messing up the dayfile.

At the end of the job, the different OUTPUT files are gathered together and concatenated to one big OUTPUT file. As a last item, the contents of Q5DAYFLE gets copied there, and then everything is sent away to the front-end, for further shipping to your printer.

PART III

OPTIMIZATION

SCALAR PROCESSOR



LSI Scalar Processor Block Diagram

12.0 THE SCALAR PROCESSOR

12.1 INTRODUCTION

The CYBER 203 and the CYBER 205 each have two distinctly different central processors: one for scalar and one for vector operations. Even such a fundamental characteristic of a computer as the internal unit of time, the clock cycle, is independently defined for the two types of processors: The CYBER 203 has a 20 ns (nanosecond) scalar and a 40 ns vector cycle length, while on the CYBER 205 both are 20 ns. The scalar processors are, for all practical purposes, identical on the two machines. Another distinction concerns the workspaces utilized: the scalar processors use mainly the register files, while the vector processors mostly deal with main memory. As a result of these and other differences, there are two distinctly different aspects of the topic of performance improvement of your code: scalar optimization, to which chapters 12-14 are devoted, and vectorization, which is treated in chapters 15-17.

Before we go on, we must define what scalar processing really is. Assuming that you are familiar with some "conventional" scalar computer, that is a comparatively easy task. The CYBER 203 (CYBER 205) scalar processor can namely do anything that you have become used to expect from a "conventional" computer: issue instructions, perform integer and floating-point arithmetic, perform logical operations, branch from one address to another, plus a few other things. We could actually totally remove the vector box, i.e. the physical part of the machine that houses the vector processor, and still have a very good and functional computer. For the purposes of the discussion in this chapter, we will define a scalar instruction as an instruction that does not utilize the vector processor for its execution.

It will prove convenient to perceive the scalar processor as consisting of a central, organizing part, surrounded by several functional units (arithmetic units, branch units etc.). The central part deals with the locating, fetching and issuing of instructions, thereby using and maintaining an instruction stack, as discussed in section 12.3. The functional units are

12.Ø THE SCALAR PROCESSOR
12.1 INTRODUCTION

responsible for the actual execution of the instructions, and will be discussed in section 12.4.

The material in this chapter is presented with the objective of providing you with enough of an understanding of the scalar processor to enable you to write good scalar code for the CYBER 2Ø3 and the CYBER 2Ø5. The same concepts can of course also be applied towards hand optimization (rewriting) of already existing code. If you feel that the content is too technical in nature, you may well skip the rest of the chapter, and concentrate on the actual examples given in chapter 13.

12.2 THE REGISTER FILE

The register file is a set of 256 64-bit working registers, serving as a work space for the scalar processor when it executes instructions. The source operands for any scalar instruction (except LOAD) come from the register file, and that is also where the result ends up (except for STORE). References to the main memory are (in scalar mode) made exclusively by LOAD/STORE instructions: LOAD fetches a value from memory and places it in the register file, and STORE does the opposite.

By convention, 15 registers are reserved for special purposes, and 17 serve as temporary registers, i.e., they are not expected to be preserved across subroutine calls. When a particular subroutine is entered, an image of all registers of importance to the caller will be saved in memory. That frees up 224 registers that can be used by the callee until it's time to return to the caller. Just before the return, the previously saved image of the register file will be loaded back into the appropriate slots, so that the caller will not notice any change of the register contents. We will return to this topic in the next chapter.

The compiler, in compiling a given subroutine, say SUBA, reserves some (maybe a few dozen) registers for purposes of holding addresses and intermediate results. But the largest part of the register file is used to hold SUBA's local, scalar variables. Each such variable gets assigned one register, and we can think of it as if the register was named after the variable. Scalar means in this context non-subscripted, and local means that it's neither part of any COMMON block, nor passed to another routine as a parameter. In the code below, Y, B, N, F, G are scalar, but only Y and G are local and scalar.

12.Ø THE SCALAR PROCESSOR
12.2 THE REGISTER FILE

```
      SUBROUTINE SUBA (X,Y)
      DIMENSION X(1ØØ)
      COMMON /BLK1/ A(1Ø),B,C(5),N
      F = X(N) - X(1)
      G = MYFUNC(F)
      Y = A(3)**2 + G*B
      RETURN
      END
```

For a subroutine that contains more local, scalar variables than there is room for in the register file, the above scheme will clearly not work. In that case some variables will be assigned memory locations rather than register file slots. Local, scalar variables appearing in EQUIVALENCE statements or function/subroutine references, or that belong to COMMON blocks, are also forced to memory. Exceptions are the arguments to most SYSLIB (system library) functions.

One of the options on the FORTRAN control card is M. If chosen, the compiler will produce a register file map, which will tell you which variables in any one subroutine that were assigned a register, and which were not. That may sometimes be helpful when you are trying to find out which subroutines make efficient use of the register file and which would perform better if split up into more than one chunk.

The importance of having scalar variables permanently assigned a slot in the register file is a consequence of the fact that a LOAD takes a comparatively long time, namely 15 cycles, to complete. If the variables A and B, but not C, have a slot assigned there, then A=C+1.Ø takes 15 cycles longer to execute than A=B+1.Ø, since C has to be loaded from memory before the addition can take place. Fortunately the LOAD instruction takes only 1 cycle to issue, and while waiting for the result to arrive to the register file, other useful work can in general be performed - but maybe not enough to completely bury the load time. Some more illuminating examples will be given when we have discussed the functional units. It should be stressed though, that to have frequently used scalar variables, such as loop indices, in memory, may well make your code run noticeably slower than it would if they had a slot in the register file.

12.3 THE INSTRUCTION STACK

The set of instructions constituting the machine code representation of your program is stored prior to execution in the code section of the controllee file. The "central part" of the scalar processor is responsible for deciphering and issuing those instructions, while the actual execution is handled by an appropriate functional unit (c.f. section 12.4). We have previously learned that nothing can really be read while remaining on disk, so the first step towards processing must clearly be to move a page of code into memory. That is however not sufficient to make the code interpretable. Just like the functional units operate on the register file, the "central part" operates on an instruction stack, i.e. it can only read and decipher instructions residing there.

The size of the instruction stack is 8 swords, where "sword" is a contraction of "super word", a term used to describe 8 consecutive words in memory. All swords start on sword boundaries, i.e. the address of a sword is always a multiple of 8*64 = 512 = #2Ø0 (bits). Instructions are not loaded from memory one by one, but rather in units of 1 sword. In addition, there is a look-ahead feature that tries to maintain the content of the instruction stack two full swords ahead. To be more precise: whenever the first instruction in sword K is read, a load request for sword K+2 is issued, provided that sword is not already in the stack.

Thanks to the look-ahead mechanism, the processing of sequential code always proceeds smoothly, with no extra delays for the loading of new instruction swords. If, however, a branch instruction is encountered, which points to a part of the code that is not currently in stack, then the appropriate sword must be loaded and a delay of 15-18 cycles is incurred. Given the information that an "in-stack" branch takes 8-9 cycles, we see that an "out-of-stack" branch takes about 3 times as long. Note that "in-stack" branches can go both forwards and backwards. It's also helpful to perceive the stack as being cyclic: when sword K is loaded, it will in general overwrite sword K-8.

A do-loop represents a piece of (approximately) sequential code, which is traversed from top to bottom, and then reentered at the top by means of a backward branch. If the branch instruction is part of sword K, then swords K+1 and K+2 are already loaded when the jump is about to take place. Hence the lowest numbered sword

present in the stack is sword K-5, and, if the branch should be an "in-stack" branch, the total span of the do-loop must not exceed 6 swords. Scalar instructions, excluding branches, each require one halfword of storage. Branch instructions and most vector instructions need a full word. Hence a sword contains 8-16 machine instructions. The largest loop that is guaranteed to fit in stack is five full swords plus a halfword long, which amounts to 81 half words or 40-80 instructions. (82 half words will not fit when the first instruction is in the end of a sword).

For a programmer who is concerned with even minute performance improvements, the question of whether a given do-loop, or for that matter a small subroutine, will fit in stack or not, may sometimes be of interest. On a machine with a small instruction stack that is indeed a relevant concern. However, with a stack size of 50 instructions or more, the potential gain from such considerations is in general negligible, and that particular issue does not deserve much attention. Moreover, as we will find later, longer loops are in general more efficient than short ones - for other reasons.

12.4 THE FUNCTIONAL UNITS

The issuing of an instruction, which includes reading it, deciphering it and feeding it to an appropriate functional unit, takes exactly 1 cycle in almost all cases. The only important exception is STORE, which issues in 2 cycles. When a given instruction has been issued, the central part of the scalar processor is free to start the process of issuing a new one, and we can thus ideally obtain an issue rate of 1 instruction per cycle. If that was the whole story, and also if each instruction produced one result, then we could classify the CYBER 203 and the CYBER 205 as 50 mflops (megaflops) machines in scalar mode. The unit mflops stands for "millions of floating-point operations per second" and has become the standard speed unit for super computers.

In reality the average number of cycles between the issue of two consecutive scalar instructions is somewhat higher than 1 - maybe 2-5 for well written codes, and even higher otherwise. Many factors are responsible for that slowdown, and a couple of the important ones will be mentioned in this section. The discussion will neither be complete nor very detailed, since the objective is to enhance your FORTRAN programming skills rather than to make

you an expert in high performance assembly language coding.

Branch instructions, which are processed by the branch unit, make decisions about where to jump, i.e. which instruction to execute next. Clearly it doesn't pay to even look at the next instruction until such a decision, if pending, is made - and that takes 8 cycles. Should the branch also be an "out of stack" branch, then an additional delay of 15-18 cycles will be incurred, as previously discussed. In this model an actual "in-stack" branch requires 1 cycle, while a "fall-through" (no branch) takes no time at all. So we can summarize by saying that a branch instruction takes 8-26 cycles to issue, and Ø-1 cycle to execute.

The L/S (load/store) unit handles memory references, i.e. LOAD and STORE instructions, as discussed in section 12.2. A LOAD issues in 1 cycle, but the loaded value is not available in the register file until 14 cycles later. A STORE, which moves the content of a register to memory, takes 2 cycles to issue, and an additional 8 cycles to complete. For a FORTRAN programmer, the relatively long load time is the most important thing to consider, and some examples to illuminate that fact will be presented in the next chapter.

The (scalar) computational work is performed by the four arithmetic units:

Type of Unit	Type of Operations	Length
Integer	Integer +,-	1
Floating point	Floating point +,-,*	5
Divide	Floating point divide, square root	54
Logical	Logical operations	3

If a given arithmetic instruction X is issued at cycle time t, then a subsequent instruction Y, needing the result of X, can issue at cycle time t+L, where L is the length of the appropriate unit as given above. If Y does not need the result of X, then Y can issue at cycle time t+1, since all arithmetic instructions have a 1 cycle issue time. A special situation occurs when the divide unit is involved, since that unit is busy for 5Ø cycles after a pair of input operands are accepted. Thus a divide or square root instruction issued at time t, will prohibit any other instruction directed to the divide unit from being issued until time t+5Ø.

The things to remember from this section are:

1) An instruction requiring an input operand from memory rather than from the register file must be preceded by a LOAD, causing a delay of up to 15 cycles.

2) An instruction that needs the result from a previous instruction cannot issue until that result is available. That will in general cause a delay of 0-4 cycles, but if the awaited result is computed in the divide unit, up to 53 cycles is possible.

3) Instructions using the divide unit must be separated with a minimum of 50 cycles. If necessary, the issues will be delayed to satisfy this.

4) Branch instructions consume a disproportionate amount of (issue) time: 8-27 cycles.

5) Items 1-4 above constitute the major (but not all) exceptions to the rule that the scalar instructions can issue (and produce results) at a rate of 1 per cycle (50 mflops).

13.Ø SCALAR OPTIMIZATION

13.1 INTRODUCTION

Assuming that your code, which previously was used in production runs on some other machine, now is in satisfactory shape for the CYBER 2Ø3 and the CYBER 2Ø5, the time has come to consider performance improvements. Several approaches are available to you, but the ones that can be described as "automatic" will probably attract you the most. It's important to realize though, that no optimizer in the world can convert a section of poorly written code to a masterpiece. The quality of the programmer is just as important as that of the optimizer. You are, therefore, urged to take an active interest in optimization - only when man is cooperating with machine can the optimum performance be achieved.

The ideas expressed in this chapter will to some extent be based on materials developed in the previous chapter. However, if you are not interested in the underlying technical concepts, you may still proceed from here, in spite of the fact that you skipped the last chapter. Chances are that you will benefit from the material below anyway.

13.2 AUTOMATIC OPTIMIZATION

At compile time, the user has the option of selecting one of several types of automatic optimization. The last parameter on the FORTRAN card is "O=...", and using that is how you specify your choice(s). The letters O, E, Z, R and I are relevant to scalar optimization, while U and V concern vectorization. The latter two will be discussed in a subsequent chapter.

Three distinctive types of (scalar) optimization exist: Z, R and I. To obtain them all (plus a little extra) the option O should be chosen (O=BO), while the letter E effectively selects R+I. Although the automatic optimization efforts are going to be

13.Ø SCALAR OPTIMIZATION
13.2 AUTOMATIC OPTIMIZATION

expended by the compiler, it might be interesting to know something about what the letters really mean. It's also easier to help the compiler (by writing better code) if you know a little about its methods.

The R-option requests "redundant code removal" which essentially amounts to recognizing and removing the computation of already computed quantities. An example would the expression X=B/(C+D) followed by Y=R/(C+D), where the sum only needs to be computed once.

The Z-option moves loop independent code out of do-loops. It also speeds up subscript calculations in loops, by treating multi-dimensional arrays as one-dimensional entities. A few other types of loop optimizations are also part of Z.

The I-option is mainly performing instruction scheduling. Some examples would be: separate divide instructions as much as possible; move loads to as early a point in the code as possible, so that the loaded values are more likely to be available when needed; move stores down and away from the instruction producing the result to be stored; separate (in an appropriate way) instructions where a later instruction needs the result of a previous one.

The first natural step after you have completed a successful test run using O=B, or no option at all, would thus be to recompile with O=BO and make a new run. If everything works smoothly, i.e. if no compilation problem occurs and if the new test results coincide with the ones initially produced, then you are obviously home free. And chances are that your SBU-number (cost) decreased quite a bit. However, life is sometimes tough, and maybe you are in bad luck. If such is the case, you may want to take one or several of the following actions.

If the compiler fails in a particular subroutine, then your CDC analyst (or salesperson) is the right person to contact and dump the problem on. He or she will most likely submit a PSR (problem report) and you can take a vacation knowing that your problem is in good hands. Should you, however, be more interested in obtaining useful results than in perfecting the compiler, then you might want to turn to other life-sustaining activities. One such would be to split off the troublesome subroutine from the rest, and compile it separately and unoptimized. It could then be merged back in binary form at load time (a list of up to 1Ø

binary files can be specified on the LOAD card) or prior to that, using OLE. Another remedy might be to compile all routines with E instead of O, or, if that doesn't work either, with some other combination of Z, R and I. Two examples of the split/merge method follows below. If a lot of computation will be done in subroutine X, then it may of course be worthwhile to try to compile X with E, Z, R or I instead of just plain nothing.

```
FORTRAN,B=B1,O=BO.
FORTRAN,B=B2.
LOAD,B1,B2,....
.
.
.
.
7/8/9
   All routines except X
7/8/9
   Subroutine X only
6/7/8/9
```

```
FORTRAN,B=B1,O=BO.
FORTRAN,B=B2.
OLE,I=B1,B2,M=B12.
LOAD,B12,...
.
.
.
7/8/9
   All routines except X
7/8/9
   Subroutine X only
6/7/8/9
```

Note that optimizers - on the CYBER 203 and the CYBER 205 as well as on other machines - are very sophisticated creations, and as such may sometimes go a little bit too far in their effort to produce optimal code. It's therefore important to compare the results generated by the unoptimized and the optimized code. If there is any discrepancy, then you must either abandon the particular optimization option chosen, or try to isolate the subroutine(s) in error.

13.3 GENERAL TECHNIQUES

a) Variable initialization is more effectively taken care of by data statements, which are satisfied once (by the loader), than by assignment statements, which are executed every time a particular subroutine is entered. This represents a small effect, not specific to the CYBER 203 or the CYBER 205.

b) Avoid double-precision calculations - they are about 10 times as expensive as single precision. Moreover, they are in general not justified, since the CYBER 203 and the CYBER 205, with their 48-bit coefficient fields, can represent floating-point numbers with about 14 significant decimal digits in single precision mode - more than enough for most practical purposes. The same considerations are valid on

60-bit CYBER-machines (like CYBER 74, 175, 176 etc.). As we shall see later though, the CYBER 203 hardware allows certain operations, namely vector sum and vector dot product, to be performed in double precision mode - at single precision speed!

c) Avoid EQUIVALENCE statements, since their existence has a detrimental effect on the compiler's ability to do a good optimization job. That's in general true for other machines as well. On the CYBER 203 and the CYBER 205, however, there is the additional aspect of the register file: equivalenced scalar variables are forced to memory, i.e. the load/store overhead will increase significantly. The equivalencing of arrays with each other is not as serious as when scalar variables are involved, and even then we are in general talking about a small to moderate effect.

d) Scalar variables should preferably not be kept in COMMON blocks, since that would increase load/store overhead.

e) IF-statements are comparatively time consuming, and their use should be kept as low as possible. An intrinsic function, such as AMAX1, AMIN1 and SIGN, can often be used in place of one or several IF-tests, and thereby bring the execution time down. That's often true on other machines as well.

f) A computed GOTO statement does not become favorable to use on the CYBER 203 or the CYBER 205 until the number of possible places to branch to exceeds 4. For 4 choices or less, a string of IF-tests is cheaper - in particular since the individual tests can often be ordered in such a way, that the branch is more often taken early than late.

g) Factorize arithmetic expressions. Consider, as an example, the evaluation of the polynomial Y=A+B*X+C*X**2+D*X**3, which requires 3 additions, 3 multiplications and 2 exponentiations. By rewriting the statement as Y=A+X*(B+X*(C+D*X)) we can bring that down to 3 additions and 3 multiplications - and that means money on any computer. (Many compilers, including CYBER 200 FORTRAN, will evaluate X**2 and X**3 as X*X and X*X*X respectively.)

h) Group subexpressions together for easier compiler recognition. As an example,

```
Change from:   X = Y*Z/A
               T = Y*V/A

        to:    X = Z*(Y/A)
               T = V*(Y/A)
```

If the subexpressions appear sufficiently close to each other, then an action like the one above will be sufficient to make the compiler aware of the fact that Y/A only needs to be computed once. In other cases, however, it is recommendable to invent new scalar variables to hold the values of such subexpressions. As a rule of thumb, statements not separated by IF-statements or subroutine calls, or by statements carrying a label, could be considered being "close enough", but the concept is by necessity somewhat vague. When in doubt, invent new variables.

i) The compiler preprocesses expressions containing constants. A helping hand from the programmer can therefore save time during the execution. If the statement in question will be executed often, that type of saving may be significant. As an example,

Change from: `Y = 180.0 * ACOS(V)/3.14`

to: `Y = (180.0/3.14) * ACOS(V)`

The first form requires two multiplications at execution time (the compiler creates the reciprocal of 3.14), while the second only requires one.

j) Minimize the number of divides. If the same denominator is used in several divide expressions, it's much faster to first compute the reciprocal, and then do multiplies instead. That will cause a marginal decrease in accuracy, due to the fact that the number of arithmetic operations per division will increase from one to two. However, other sources of errors will almost always dominate, and the method of reciprocals is indeed as computationally sound. Code on almost all machines will benefit from this type of change.

k) Separate divides/square roots, so that no unnecessary delay occurs - remember that two instructions utilizing the divide unit must be issued no less than 50 cycles apart. The compiler will spread out such instructions as much as possible, but cannot cross barriers such as statement labels, IF-statements and subroutine calls - they have to be crossed by the programmer! An example would be to move one of two consecutive divides, possibly by computing the reciprocal, so that it is issued before a do-loop rather than after it. That would allow the first divide to complete while the do-loop executed, whereafter the second could be issued immediately. Similarly, if the divide or square root part of an expression could be moved up before a do-loop or some other time-consuming section of code, then the 54 cycles

needed to produce a result would effectively reduce to 1.

1) Consolidate divides and square roots wherever possible. Such changes can sometimes speed up a code section by an order of magnitude! Examples:

```
Change from:   A = B/C/D/E
               X = SQRT(F) * SQRT(G)

        to:    A = B/(C*D*E)
               X = SQRT(F*G)
```

13.4 REGISTER FILE UTILIZATION

To save on load/store overhead, intermediate results should be kept in the register file rather than in memory. The following loop is an example of poorly written code in that respect:

Example a1:

```
      DO 1Ø J=1,N
      X(J) = A(J)**2
      Y(J) = 1.  + X(J)
1Ø X(J) = T*X(J)+Y(J)
```

One of the "flaws" is the redundant store of X(J) in the first line. The introduction of a temporary scalar variable will take care of that:

Example a2:

```
      DO 1Ø J=1,N
      TEMP1 = A(J)**2
      Y(J)  = 1.+TEMP1
1Ø X(J)  = T*TEMP1 + Y(J)
```

We also notice that the loading of Y(J) in the last line is unnecessary, and could be made to disappear:

Example a3:

```
      DO 1Ø J=1,N
      TEMP1 = A(J)**2
      TEMP2 = 1.+TEMP1
      Y(J)  = TEMP2
1Ø X(J)  = T*TEMP1 + TEMP2
```

The steps above illustrate the use of the register file. In addition, they serve as examples of the efficiency of the current FORTRAN compiler (1.4): all three loops will be assembled to

logically identical codes, provided the O-option is selected. That means that they all will execute at the same speed, and that nothing was gained by the hand optimization. Sometimes, though, a loop may contain logical ambiguities that the compiler cannot resolve. In such cases a small amount of effort expended by the programmer may pay off very nicely. As an example, consider the two logically identical (?) code-sections below:

Example b1:

```
      DO 3Ø J=1,M
      DO 2Ø K=2,N
      A(J,1,1) = A(J,1,1) + A(J,K,1)
2Ø    A(J,1,2) = A(J,1,2) + A(J,K,2)
3Ø    CONTINUE
```

Example b2:

```
      DO 3Ø J=1,M
      A1 = A(J,1,1)
      A2 = A(J,1,2)
      DO 2Ø K=2,N
      A1 = A1 + A(J,K,1)
2Ø    A2 = A2 + A(J,K,2)
      A(J,1,1) = A1
      A(J,1,2) = A2
3Ø    CONTINUE
```

When O=BO is selected, the FORTRAN 1.4 compiler generates code with the following timings:

$$t(b1) = (25+50N)M \text{ cycles}$$
$$t(b2) = (30+17N)M \text{ cycles}$$

The programmer introduced utilization of the register file thus cut the execution time by a factor of 3 for large values of N! That could not have been done automatically by the compiler since the code (b1) is actually ambiguous. Suppose, namely, that A is dimensioned A(1ØØ,1Ø,3) and that N=11. Then the last pass through the inner loop in (b4) will correspond to executing the following two statements:

```
A(J,1,1) = A(J,1,1) + A(J,11,1)
A(J,1,2) = A(J,1,2) + A(J,11,2)
```

By applying the formula given in the FORTRAN manual, page 2-3, you can now convince yourself that A(J,11,1) and A(J,1,2) actually correspond to the same memory location; they are both the (J+1ØØØ)th element of array A. The step above will thus add to A(J,1,1) the value of the accumulated sum A(J,1,2). However,

the corresponding step in (b2) will add the old, unmodified value of A(J,1,2), since the accumulated sum there is kept in A2 rather than in A(J,1,2). Thus, in this special case (b1) and (b2) give different results. You may very well be a nice person that never would even dream of exceeding the limits specified in the dimension statement, but the compiler doesn't know that, and consequently refuses to translate (b1) into (b2).

13.5 RECURSIVE DO-LOOPS

If the Jth pass through a do-loop uses results calculated in pass J-1 (or earlier), then the loop is said to be recursive. Such loops can often be speeded up significantly by keeping duplicates of some values in the register file. A typical example is the computation of the factorial of a number: F(J) = (J-1)!. The naive approach would be to write the following loop [NM1 = N-1]:

Example c1:

```
      F(1) = 1.
      F(2) = 1.
      DO 10 J=2,NM1
10    F(J+1) = J*F(J)
```

A moment's thought reveals that the F(J) that is stored during pass J must be loaded again during pass J+1. So why not keep the value in the register file, and avoid the load?

Example c2:

```
      F(1) = 1.
      F(2) = 1.
      FACT = 1.
      DO 10 J=2,NM1
      FACT = FACT*J
10    F(J+1) = FACT
```

The high performance programmer may want to go one step further, and do a little manual instruction scheduling:

Example c3:

```
      F(1) = 1.
      F(2) = 1.
      FACT = 2.
      XVAL = 3.
      DO 10 J=3,N
      F(J) = FACT
      FACT = FACT * XVAL
10    XVAL = XVAL + 1.
```

The difference between (c2) and (c3) appears to be that a given pass through the loop in (c2) requires the execution of three recursive instructions (FLOAT, MULTIPLY, STORE), while in (c3) there is no recursion within a pass. To execute a sequence of recursive instructions is in general slower, since the issue of instruction K has to be delayed until the result of instruction K-1 is available, and we would therefore expect (c3) to be significantly faster. However, with a good compiler, that's not necessarily true, as the following timings show [FORTRAN 1.4, O=BO selected]:

t(c1) = 18 + 29N cycles
t(c2) = 18 + 15N cycles
t(c3) = 9 + 14N cycles

A factor of 2 was thus gained by going from (c1) to (c2), while the change to (c3) had almost no effect. That does not in any way invalidate the ideas expressed above, but rather tells us that the compiler also can think - and that's reassuring. Had the original loop been a little more complicated, it might well have proven to be too difficult for the compiler to find the optimum way of assembling the code. In that case the optimization steps corresonding to c1-c2 and c2-c3 would both have been very profitable.

As an additional example of recursive loops, let's consider the calculation of the elements in a Fibonacci series, which are defined by:

F(1) = F(2) = 1.
F(J) = F(J-1) + F(J-2) (J.GE.3)

The straightforward way of coding that would be the following:

Example d1:

```
      F(1) = 1.
      F(2) = 1.
      DO 1Ø J=3,N
   1Ø F(J) = F(J-1) + F(J-2)
```

But by saving the values of the two lastly computed elements in the register file, we can create a much more efficient version:

Example d2:

```
      F(1) = 1.
      F(2) = 1.
      FJM2 = 1.
      FJM1 = 1.
      DO 10 J=3,N
      FJ   = FJM1 + FJM2
      FJM2 = FJM1
      FJM1 = FJ
   10 F(J) = FJ
```

The timings are as follows [FORTRAN 1.4, O=BO]:

t(d1) = 18 + 30N cycles
t(d2) = 16 + 16N cycles

Also in this case the payoff was thus a reduction in execution time by a factor of 2. No additional speedup can be achieved without unrolling the loop (section 13.7).

13.6 THE MERGING OF SHORT DO-LOOPS

A load-bound loop is a loop that is dominated by the 15 cycles needed to complete a load instruction, and which would not run significantly faster even if all other instructions executed in zero time. Similarly, a branch-bound loop is dominated by the 9 cycles needed to do a "test and branch" in the end of each pass.

Busy loops, i.e. loops in which a lot of work is performed during each pass, are never branch-bound. In general, they are not load-bound either, since load instructions often can be issued early, permitting useful work to be done while waiting for the loaded values to arrive to the register file. Even disregarding the loads, the automatic instruction scheduling becomes more efficient when the loop is busy; the more instructions the scheduler can shuffle around, the better the results that can be achieved.

In contrast, very short do-loops are almost always either branch-bound or load-bound. A way to capitalize on that fact is to merge small loops into bigger ones - an action that most of

the time will boost scalar performance significantly. It's worth noting though, that if the small loops are automatically vectorizable, and if you plan to select the V-option at compile time, then the merging into bigger loops won't buy you anything - but it won't harm you either. [Refer to subsequent chapters for vectorization techniques]. Examples:

Change from e1:

```
      DO 1Ø J=1,N
1Ø X(J) = A(J)**2 + B(J)**2
      DO 2Ø J=1,N
2Ø Y(J) = R(J) + S(J)
```

to e2:

```
      DO 1Ø J=1,N
      X(J) = A(J)**2 + B(J)**2
1Ø Y(J) = R(J) + S(J)
```

With O=BO selected, the FORTRAN 1.4 compiler generates code that executes as follows:

t(e1) = 31 + 37N cycles
t(e2) = 16 + 25N cycles

When N is large, this particular merging thus results in a 32% performance improvement. Such a gain is typical for similar actions.

13.7 THE UNROLLING OF DO-LOOPS

When merging is not feasible, you may want to consider the unrolling of a short loop to 2 or more levels. That will also reduce loop overhead and give the instruction scheduler more material to work with; performance improvements similar to those obtained from merging can therefore be expected. There is, however, one significant difference: by unrolling an automatically vectorizable loop, the vector property is always destroyed. As pointed out in the previous section, the merging of loops does not have that effect. In addition, it's usually a little messier to unroll a loop, because end cases will appear that must be taken care of. The examples below illustrate the unrolling of a loop to 2 and 3 levels respectively.

13.0 SCALAR OPTIMIZATION
13.7 THE UNROLLING OF DO-LOOPS

Example f1:

```
      DO 10 J=1,N
10    A(J) = X*B(J) + C(J)
```

Example f2:

```
      IF (N.EQ.1) GO TO 11
      NM1 = N-1
      DO 10 J = 1,NM1,2
      JJ = J
      A(J) = X*B(J)     + C(J)
10    A(J+1) = X*B(J+1) + C(J+1)
      IF (JJ.EQ.NM1) GO TO 12
11    A(N) = X*B(N) + C(N)
12    CONTINUE
```

Example f3:

```
      IF (N.GT.2) GO TO 9
      IF (N.EQ.1) GO TO 12
      GOTO 11
 9    NM2 = N-2
      NM3 = N-3
      DO 10 J = 1,NM2,3
      JJ     = J
      A(J)   = X*B(J)   + C(J)
      A(J+1) = X*B(J+1) + C(J+1)
10    A(J+2) = X*B(J+2) + C(J+2)
      IF (JJ.EQ.NM2) GOTO 13
      IF (JJ.EQ.NM3) GOTO 12
11    A(N-1) = X*B(N-1) + C(N-1)
12    A(N)   = X*B(N)   + C(N)
13    CONTINUE
```

The timings are as follows [FORTRAN 1.4, O=BO selected]:

```
t(f1) = 14 + 19.00N cycles
t(f2) = 52 + 12.50N cycles
t(f3) = 76 + 11.33N cycles
```

Obviously the greatest gain (34%) was obtained by unrolling to 2 levels, and, in light of the increased complexity, it does not seem justified to go beyond that. Typically you may expect 30-40% performance improvement when unrolling to 2 levels, and 40-50% when unrolling to 3. Note that longer loops would not benefit as much from unrolling.

13.8 THE SPLITTING OF DO-LOOPS

A do-loop containing one or several loop-independent IF-tests will, in general, execute quite a bit faster if split into smaller loops, as in the following example:

Change from g1:

```
      DO 1Ø J=1,N
      A(J) = X(J)**2 + Y(J)**2
      IF (IFLAG.EQ.Ø) A(J) = A(J) + DELTA
1Ø    CONTINUE
```

to g2:

```
      DO 1Ø J=1,N
1Ø    A(J) = X(J)**2 + Y(J)**2
      IF (IFLAG.NE.Ø) GOTO 12
      DO 11 J=1,N
11    A(J) = A(J) + DELTA
12    CONTINUE
```

Clearly (g2) must be faster when IFLAG=O, since then much less code is executed in (g2) than in (g1). But even when IFLAG=O the splitting is advantageous. Noting that there are 2N branch instructions to be executed in (g1), and 2N+1 in (g2), that may at first sound a little bit surprising. As it turns out, though, a more important difference than the number of branch instructions is the fact that the loop in (g1) contains an IF-statement, while those in (g2) don't. The two loops in (g2) can therefore be better optimized by the compiler - an IF-statement in the middle seriously affects the efficiency of the instruction scheduling. In particular, the loops in (g2) will both be assembled with "top store - bottom load", while the one in (g1) will not. The timings are as follows [FORTRAN 1.4, O=BO selected]:

```
t(g1) = 1Ø + 59N cycles    (IFLAG.EQ.Ø)
t(g2) = 38 + 46N cycles    (IFLAG.EQ.Ø)

t(g1) = 1Ø + 5ØN cycles    (IFLAG.NE.Ø)
t(g2) = 26 + 17N cycles    (IFLAG.NE.Ø)
```

Thus, for large values of N, the splitting gains us 66% if IFLAG=Ø, and 22% otherwise. It's also worth noting that the loops in (g2) will vectorize automatically, if the V-option is selected at compile time, while the one in (g1) won't. That will be further clarified in the chapters about vector processing.

14.Ø HOW TO SPEED UP SUBPROGRAM CALLS

14.1 INTRODUCTION

A program usually consists of several independent program units: one main program and several subprograms (subroutines and/or functions). The interfaces between these units are handled by FORTRAN executable CALL statements, function references and RETURN statements, and essentially consists of the transfer of control (jumps or branches) in conjunction with the exchange of information (parameter passing). The usage of subprograms gives the programmer several advantages: fewer lines of code are needed, which in turn saves on punch-cards and eliminates some potential for typing errors; legibility is increased (the logic is easier to follow); debugging is facilitated; the code is easier to maintain; etc.. There is one drawback, though: a section of code, when split off and transformed to a subprogram, will execute slower than it did before. That is due to the fact that a certain amount of overhead is incurred every time that newly created routine is called. For routines that perform a lot of work, the time spent doing "useful" work will dominate, and the question of overhead can be ignored. However, repeated calls to a subprogram that performs a trivial task may well cause the overhead associated with those calls to show up as a significant part of your total job-cost.

The two dominating contributions to the overhead are register file swapping and parameter transfer. The swap-concept will be explained in the next section, and there we will also show how swaps might be avoided under certain conditions. In the section after that we will explain how the transfer of parameters works, and that will hopefully provide you with an understanding for why short parameter lists sometimes can be a lot cheaper than long ones. The remaining sections of this chapter contain examples of different ways to improve performance by reducing the number of subprogram calls.

14.2 REGISTER FILE SWAPPING

When compiling a particular subroutine, say SUB1, the compiler generates code which, assuming that neither option O or Z is specified, has the following structure [minor bookkeeping tasks are omitted]:

1) Prologue: Swap out (save) caller's registers.
   Swap in SUB1's scalar, local variables.
   Load dummy parameters.

2) Compiled FORTRAN statements.

3) Epilogue: Store dummy parameters.
   Swap out SUB1's scalar, local variables.
   Swap in (restore) callers registers.
   Return to caller.

At execution time, when SUB1 gets control, the execution of the calling routine has just been interrupted, and a branch to SUB1 has taken place (a CALL-statement has been executed). At that time, the contents of the register file is associated with the caller; in particular, the values of the caller's local, scalar variables are stored in registers. SUB1's first task must therefore be to save an image of the caller-utilized part of the register file somewhere in memory. A machine instruction called SWAP is designed for just that purpose, and the procedure of saving registers is therefore often referred to as "swapping out".

Chances are that there are a lot of scalar, local variables that got values assigned to them during the previous pass through SUB1, or, if this is the first time SUB1 is called, still have the values specified in the DATA-statements. These variables are kept stored in memory (in SUB1's data base) when SUB1 is not executing. At this time, though, when SUB1 has been entered and the caller's registers have all been saved, they can be brought up and stored into preassigned register locations. Thus, during the second phase of the prologue SUB1's scalar, local variables are swapped in from memory to the freed-up register file. From then on, until it's time to consider returning to the caller, SUB1 can utilize the register file for its own purposes, namely as a work space for the tasks it was designed to perform (arithmetic calculations, etc.).

14.Ø HOW TO SPEED UP SUBPROGRAM CALLS
14.2 REGISTER FILE SWAPPING

Fortunately, the SWAP is "two-sided", i.e., it can both swap in and out at the same time. Only one machine instruction is thus needed to accomplish the saving of the caller's registers and the filling of the register file with the scalar part of SUB1's data base. That's the good news. The bad news is that it takes a comparatively long time to do a swap: N registers are swapped in about 14Ø+N cycles (2Ø ns), which evaluates to 5-7 microseconds. Since, as part of the epilogue, one more two-sided SWAP has to be performed in order to restore the register file and save SUB1's variables, we are talking about 1Ø-15 microseconds of overhead just from the SWAPs.

As indicated in section 12.2, there are 17 temporary registers which always are available, i.e., the caller cannot expect them to be preserved over a subprogram call. A routine designed to do only a small amount of work (just a few lines of FORTRAN) can often perform its task efficiently without utilizing the remaining registers, and would thus not benefit very much from swaps. Moreover, since very little work (besides overhead) would be performed by such a routine, its execution time would be totally dominated by the swaps - if present. In contrast, the lack of register space would cause most moderately-sized subroutines to perform very poorly, should the swaps be omitted. But for those routines, there is really no good reason to try to get by without swaps in the first place, since the swap-time is trivial compared to the time spent doing other useful work.

Swap-routines are the standard output from the FORTRAN compiler. Under certain circumstances, however, zero-swap-routines are generated - usually leading to significant savings when they later are executed. Such routines are simply characterized by the fact that they do not contain any SWAP-instructions; they thus execute using only temporary registers. A set of sufficient conditions for zero-swap is the following:

1) Option O or Z was specified at compile time.

2) There are no calls or function references other than to FORTRAN routines that can be generated in-line [FORTRAN, Appendix E].

3) There are no INPUT/OUTPUT statements.

4) There are no explicit vector statements.

5) There are no vector instructions generated by the automatic vectorizer.

6) There are no special calls [CALL Q8...].

7) The code can be reasonably executed using only 17 (or possibly 2Ø) working registers.

14.3 PARAMETER PASSING

The only practical way of passing an array in a subprogram call is to pass the base address of the array, i.e., the address of the first element. On the other hand, a scalar variable, say X, can be passed in either one of two different ways: by value or by address. The former method consists of storing the value of X in some predetermined register, where it is immediately available to the callee. The latter method requires that X first be stored somewhere in memory, whereafter the address of that location is passed.

Although the method of calling by value appears to be simpler (it does indeed require less overhead), it does have certain drawbacks; the most important one is that it limits the number of scalar parameters that can be passed. With the exception of some calls to system library (SYSLIB) routines, all subprogram references are therefore of the type "call by address".

As far as the parameter passing is concerned, the caller has to go through the following steps of preparation to effectuate a call by address:

14.Ø HOW TO SPEED UP SUBPROGRAM CALLS
14.3 PARAMETER PASSING

1) Reserve table space in memory - 1 word per parameter to be passed.

2) Store the scalar variables in the parameter list somewhere in memory (usually in the caller's database).

3) Fill the table entries with the addresses of the scalar variables and the addresses of the first elements of the arrays. The addresses should be stored in the table in the same order as the parameters occur in the parameter list.

4) Load a special register (number #17) with the address of the first word of the table.

5) Jump to the subroutine.

When the callee gets control, the table address is thus available in register #17. If we consider the table as an array ITAB in memory, then register #17 contains the address of ITAB(1), while ITAB(K) contains the address of argument K. In particular, if the scalar variable X was passed as argument 4, then X could be loaded into the register file by first loading the value IADD=ITAB(4) into some temporary register, and then the value at memory location IADD into register file location X. To load a scalar dummy parameter thus requires two loads, which is usually referred to as a "double fetch". The same mechanism must, of course, be implemented when the value of an array element is desired, but since the base address (the address in the table) only needs to be loaded once and then can be used to access each element of that array, the average work per element is about half that required for a scalar variable.

Another way of passing parameters is of course to use COMMON-blocks. An advantage of this method is that it allows a given parameter to be accessed with only a single fetch - which obviously is a little quicker than a double. The reason why only one fetch is required is that the compiler generates code that effectively treats the whole block as one long array. One base address therefore becomes sufficient to access any location in the block - be it a scalar variable or an array element. As an example, the declaration

```
COMMON /BK/ A(1Ø),B(5),KA
```

will cause the compiler to reference B(2) as A(12) and KA as A(16).

14.0 HOW TO SPEED UP SUBPROGRAM CALLS
14.3 PARAMETER PASSING

As usual, the overhead associated with a particular process could (and should) be ignored whenever the "useful" work dominates. Therefore, we will never be concerned with the advantage of one way of passing parameters over the other when the subprogram in question is moderately sized and/or performs a significant amount of work. However, for very small subprograms, the question may be crucial. Consider, as an example, the following two cases:

Example a1:

```
      DO 10 J = 1,N
      A(J) = XNORM(X(1,J),X(2,J),X(3,J))
   10 CONTINUE
      .
      .
      FUNCTION XNORM (X,Y,Z)
      XNORM = SQRT (X**2+Y**2+Z**2)
      RETURN
      END
```

Example a2:

```
      DO 10 J = 1,N
      A(J) = XNORM (X(1,J))
   10 CONTINUE
      .
      .
      FUNCTION XNORM(X)
      DIMENSION X(3)
      XNORM = SQRT(X(1)**2 + X(2)**2 + X(3)**2)
      RETURN
      END
```

The loop in (a2) executes 20% faster than that in (a1), which, in some circumstances, may be considered significant. Another game to play with long argument lists to small subroutines is to invent a common-block especially for parameter passing, storing all scalar variables there prior to the call, thus passing only arrays as dummy parameters. Although the method is fairly messy to implement, it will buy you some cycles. Whether to use it or not is all a matter of how greedy you are - and how much your time is worth.

As a final note on this topic, it's worth mentioning that small subroutines often can benefit from being converted to functions. The reason is that the function return value is passed "by value" (i.e. in a register) rather than "by address", and therefore is accessible to the caller immediately upon return. Double precision and complex-valued functions return results in two registers.

14.4 PULL OR PUSH SUBROUTINES

The most obvious way to cut down on subprogram overhead is to reduce the number of calls or references. One way to accomplish that is to bring ("pull") the whole subprogram in line, i.e., to absorb the code into that of the caller. Example (a1) would thus be transformed into:

Example a3:

```
      DO 10 J = 1,N
      A(J) = SQRT(X(1,J)**2+X(2,J)**2+X(3,J)**2)
   10 CONTINUE
```

This method, as well as the others in this section, clearly sacrifices some of the benefits associated with having the code modularized. But everything has a price, and with the information that (a3) executes in only 47% of the time required for (a1) you might well find similar actions worthwhile. It must be emphasized, though, that an essential condition for the high gain factor is that the "pulled" subprogram performs very little work.

To justify the last statement, let's consider what would have happened if the amount of "useful" work performed by XNORM had been 10 or 100 times as large. The fact that (a3) executes 53% faster than (a1), in conjunction with the observation that (a3) is all "useful" work, tells us that the subprogram overhead in (a1) is about 53%. Therefore, pulling a 10 times more productive XNORM in-line would buy us about

$$100 * 53/(53+470) = 10\%$$

Similarly, a "productiveness factor" of 100 results in a gain of only 1%. So don't bother with subprograms that perform significant amounts of work!

14.Ø HOW TO SPEED UP SUBPROGRAM CALLS
14.4 PULL OR PUSH SUBROUTINES

To make the pulling of a subprogram in-line less cumbersome, a statement function can in many cases be utilized. To use statement functions does not affect the execution time per se, but it does cut down on programming efforts. The following code executes in a manner (and time) identical to the one in (a3):

Example a4:

```
      XNORM(X,Y,Z) = SQRT (X**2 + Y**2 + Z**2)
      .
      .
      DO 1Ø J = 1,N
      A(J) = XNORM (X(1,J),X(2,J),X(3,J))
   1Ø CONTINUE
```

The method of pulling a subprogram in-line can also be reversed: the calling loop can be pushed out to the subprogram. The number of calls would thereby be reduced from N to 1, and, although a few more parameters may have to be passed, that will usually result in savings almost as large as if the pull-method had been used - provided, of course, that N is reasonably large. In our particular case, a "push" would turn out something like:

Example a5:

```
      CALL XNORM(N,A,X)
      .
      .
      SUBROUTINE XNORM (N,A,X)
      DIMENSION A(1),X(3,1)
      DO 1Ø J = 1,N
      A(J) = SQRT(X(1,J)**2+X(2,J)**2+X(3,J)**2)
   1Ø CONTINUE
      RETURN
      END
```

14.5 VECTORIZE I/O

To a FORTRAN programmer a WRITE statement looks quite simple. To the machine, however, it represents a lot of work. We have previously discussed the actual data transfer (chapter 1Ø), but there is, of course, more than that involved. It would carry too far to explain in detail which system routines were involved, and what they do, and we will here simplify the matter as follows: Each set of contiguous items in the output list of a WRITE statement represents a series of subprogram calls, while the length of a particular set of items only shows up at the level of the data transfer. A set of items is defined here as either a scalar variable, or (a part of) an array which is specified by an implied do-loop. Hence, if A is a two-dimensional array stored by columns, then A and (A(J,K),J=1,N) both specify sets of contiguous items, while (A(J,K),K=1,N) does not.

The consequences of what's stated above is that the number of subprogram calls generated by a WRITE statement can be significantly reduced by decreasing the number of sets of contiguous items. The following three examples illuminate that method [A is dimensioned A(4,4)]:

```
b1:  WRITE (7) X,Y,Z,((A(J,K),K=1,3),J=1,3)

b2:  WRITE (7) X,Y,Z,((A(J,K),J=1,3),K=1,3)

b3:  TEMP(1) = X
     TEMP(2) = Y
     TEMP(3) = Z
     WRITE (7) (TEMP(J),J=1,3),((A(J,K),J=1,3),K=1,3)
```

In (b1) there are 12 sets of contiguous items, while in (b2) and (b3) that number is reduced to 6 and 4 respectively. Timing measurements for the three cases yield 679, 453, and 38Ø microseconds respectively, which can be expressed as b1:b2:b3 = 9:6:5. With formatted WRITEs (12E1Ø.4), the execution times become 971, 712, and 627 microseconds, and the ratios 8.2:6:5.3. Obviously neither one of these two sets of ratios come all that close to the "predicted" 12:6:4, but we can at least conclude that the basic idea of decreasing the number of sets of contiguous items seems to be quite powerful as an optimization tool. In fact, that very same idea forms the basis for the vector techniques that will be presented later in this document, and the method discussed in this section is therefore often referred to as the process of "vectorizing I/O".

14.6 OTHER TECHNIQUES

Let's assume that you have the following statement inside a do-loop:

```
      DO 10 J = 1,N
      .
      .
      PI = ACOS(-1.0)
      .
      .
   10 CONTINUE
```

Incidentally, this generates pi with an accuracy of 48 bits (95 bits in double precision). The expression is clearly an invariant with respect to loop-counter, i.e., it can be moved out and executed once only before the loop. The programmer can thus easily save N-1 trigonometric function references - but can we expect the same cleverness from the compiler? The answer is no, but not because the compiler isn't smart enough - on the contrary! Just try to imagine what would happen if the subprogram ACOS had a COMMON-block that was massaged every time ACOS was called! In that case the reduction of the number of calls from N to 1 would clearly be disastrous. The conclusion is that loop-invariant subprogram calls appearing inside loops must be moved out of the loop by the programmer - the compiler cannot do it. Note also that such actions generally result in substantial savings - N-1 executions of a subprogram rarely takes a trivial amount of time.

Another money-saver, which is based on the mathematical properties of some functions, is to consolidate references whenever possible. Some examples are the following:

Example c1:
```
A = SQRT(X)*SQRT(Y)
B = EXP(X)*EXP(Y)
C = ALOG10(X)+ALOG10(Y)
```

Example c2:
```
A = SQRT(X*Y)
B = EXP(X+Y)
C = ALOG10(X*Y)
```

Each statement in (c1) contains 2 subprogram references, while those in (c2) only contain 1. Talking dollars that means that (c2) runs for half price!

15.Ø VECTOR PROCESSING

15.1 INTRODUCTION

Up until now we have ignored the one feature of the CYBER 2Ø3 that probably has contributed most to its fame: the vector processing capability. Why? Well, would you suggest that the education of pilots for supersonic airplanes started at supersonic speeds? Probably not. And although the interesting speed with respect to vector processing is the speed of light rather than that of sound, the philosophy of that analogy certainly applies.

Having studied the previous chapters you should now have a fairly good understanding of how the scalar processor operates, and how it interacts with the register file and physical memory. That in conjunction with your (similarly acquired) knowledge about files and virtual memory will now serve as the elevated launching pad from which you confidently can explore the function and usefulness of the vector processor. One special advice we would like to give you before you start: always keep in mind that the function and usefulness of any given part of a computer is highly dependent on similar characteristics of other parts, and of its interface with them - too narrow a focusing will always prevent you from utilizing the full potential.

15.2 THE DEFINITION OF A VECTOR

Programmers working with scalar code on scalar computers traditionally pay very little, if any, attention to how arrays are stored in memory. And looking at scalar FORTRAN code, it seems quite justified - nowhere does the formulation appear to be dependent on whether the memory location holding A(K) is adjacent to the one holding A(K+1). Therefore, how can a proficient FORTRAN programmer be expected to make an intelligent choice between the following two loops?

15.Ø VECTOR PROCESSING
15.2 THE DEFINITION OF A VECTOR

```
      DO 1Ø J = 1,N                DO 2Ø K = 1,N
      DO 1Ø K = 1,N                DO 2Ø J = 1,N
1Ø A(J,K) = Ø.               2Ø A(J,K) = Ø.
```

The fact is, that although the syntax of the language completely hides it, there is a significant difference in execution efficiency between the two loops on many scalar computers. If two-dimensional arrays are stored by the columns, i.e., if A(J,K) is immediately followed by A(J+1,K) in memory, then the 2Ø-loop is the faster. Similarly, the 1Ø-loop executes better on machines that implement rowwise storage (A(J,K), A(J,K+1), etc.). In short: a loop that accesses sequential storage locations in memory is always the better choice. That's particularly true on machines that, like the CYBER 2Ø3/2Ø5, feature virtual memory, since for large values of N the paging is minimized by sequential memory access.

To write scalar code for a scalar processor totally ignoring the question about storage is usually not dramatically penalized. By changing from non-sequential data access to sequential, it is realistic to expect an increase in execution speed of 2Ø-1ØØ%, but hardly more. On a vector processor, however, the difference between the two methods is indeed dramatic - speed increases of 1ØØØ-2ØØØ% as the result of going to sequential access are not unheard of. The reason why such impressive improvements are possible can be traced to the fact that a vector processor is actually <u>designed</u> to operate on sequential memory locations. That fact, in turn, provides us with a vector definition:

DEFINITION: Vector = a set of contiguous memory locations.

The above definition is appropriate for the CYBER 2ØØ series of machines, but not necessarily for other architectures. Note also that contiguity is defined in terms of virtual addresses - information about locations in physical memory is normally not accessible to the user.

It cannot be enough stressed that contiguity of data is essential to <u>all</u> vector operations on the CYBER 2Ø3/2Ø5, and that your success therefore will be very dependent on how well you can absorb and utilize that idea.

15.3 THE VECTOR PROCESSOR

For pedagogical purposes, the vector processor on the CYBER 203/205 can be perceived as consisting of segmented pipes. Each segment can only perform a small part of an arithmetic operation, so that each pair of operands has to be processed in several steps. As an example, let's consider a floating point add, which can be split up into the following 6 independent operations [don't worry if you don't understand the details - you will not be tested on this]:

1) Compare the two exponents.

2) Right-shift the coefficient with the smaller exponent.

3) Add the coefficients.

4) Count how many steps the result coefficient must be left-shifted to become normalized.

5) Normalize and adjust the exponent.

6) Transmit the result to a memory bus.

These six steps represent just one possible subdivision, and do not necessarily reflect the one actually implemented. What is important from the user point of view is that there is a certain number of steps, and that the operands have to traverse the corresponding segments sequentially. A different vector operation may utilize a different set of segments within the same pipe, some of which may be identical to some in the list above. Each pipe can therefore be thought of as containing several arithmetic units, each of which functions as an independent pipe.

The CYBER 203 has two unique pipes, differing mainly with respect to the divide function, which can only be peformed by one of them. The CYBER 205 has in its standard form two identical pipes, but a version with four identical pipes is also offered. When 2 pipes both can perform a given operation the data is evenly distributed, so that pipe 1 processes the odd. pairs of input operands, while pipe 2 takes care of the even ones. In the 4-pipe case pipe 1 handles every fourth pair, etc.. The multiple pipe configuration can thus not be used to process two or more vector instructions in parallel.

To get a feeling for how the vector processor works, let's assume that only one pipe exists, and that we want to peform the addition of two vectors, as symbolized by the following scalar loop:

```
      DO 10 J = 1,N
10    A(J) = B(J) + C(J)
```

The first step in the process will be to initiate a streaming of the elements in arrays B and C from memory to the beginning of the pipe. A certain number of clock cycles is required to transfer B1 and C1 into the first pipe segment, and that number represents the length of the input path. Arbitrarily fixing the length to 2 cycles, and assuming that the vector instruction was initiated at cycle time 1, we will have the following pictures at cycles 1, 2 and 3 respectively:

Input path	SEG1	SEG2	. . .
(B1,C1)			
(B2,C2)	(B1,C1)		
(B3,C3)	(B2,C2)	(B1,C1)	

During the next cycle, (B1,C1) will be moved into segment 2 and processed there, allowing all other pairs to advance one step, and the pair (B4,C4) to enter the input stream. With 6 segments in total, the pipe will thus not be filled until cycle 8. When (B1,C1) has passed through segment 6, the first result, A1, enters the path back to memory. If the length of the output path also is 2 cycles, the picture at cycle times 9, 10 and 11 will thus be as follows:

Input path		SEG1	. . .	SEG6	Output path	
(B9, C9)	(B8, C8)	(B7,C7)	. . .	(B2,C2)	A1	
(B10,C10)	(B9, C9)	(B8,C8)	. . .	(B3,C3)	A2	A1
(B11,C11)	(B10,C10)	(B9,C9)	. . .	(B4,C4)	A3	A2

The first result was thus stored in memory during cycle 11, and from then on the other results will follow at a rate of 1 per cycle until the last element is stored. In this particular example we can thus specify the timing as:

10 + N cycles

Two numbers are needed to describe the behavior: a startup value (here 10 cycles) which is independent of the vector length, and a stream rate which does depend on the length, N. Two pipes can process twice as many operands per cycle, which gives us N/2 instead of N for the stream rate. In particular with respect to the startup procedure, the picture given here is greatly simplified, and the actual timing for a vector add on the CYBER 203 is:

82 + N/2 cycles

The startup time for a given instruction can only be quoted as an average value, since it's affected by several "environmental" factors such as relative locations of the three vectors (two input and one output) in memory and their length modulo 64. The fluctuations are of the order of 10%. The stream rates, however, are very exact, and any deviation from quoted values will be due to inaccuracies in the measurement method.

On the CYBER 203, multiplication and division requires two passes through the pipes. Furthermore, only one of the pipes can perform division. The stream rates are thus N and 2N respectively. The clock cycle on the CYBER 203 is 20 ns for the scalar processor, but 40 ns for the vector processor. To avoid confusion, we will henceforth quote all timings in units of 20 ns cycles; that will double the values for vector operations previously given in this section. On the CYBER 205, the clock cycle is 20 ns for both processors, and the vector processor is in addition quite a bit faster for several instructions. Using the unit 20 ns cycles, the timings for the four basic arithmetic floating point operations on the two machines are as follows:

	CYBER 203	CYBER 205 (2 pipe)
Add/Subtract	165 + N	51 + N/2
Multiply	350 + 2N	52 + N/2
Divide	360 + 4N	80 + 25N/8

16.Ø AUTOMATIC VECTORIZATION

16.1 INTRODUCTION

The CYBER 2Ø3/2Ø5 offers the user two different ways of taking advantage of its vector processing capability: automatic and explicit vectorization. The automatic vectorizer is invoked by including the letter "V" in the string of options on the FORTRAN card, and can thus be regarded as a form of compiler optimization. Explicit vectorization, on the other hand, implies that the programmer makes use of a special vector syntax, available in CYBER 2ØØ FORTRAN as an extension of the language. Usage of one method neither requires nor prohibits the use of the other. This chapter will be devoted to the different aspects of automatic vectorization, while the explicit method is treated in chapter 17.

16.2 GENERAL CONSIDERATIONS

The nature of vector operations is such that the only types of code structures that can qualify for automatic vectorization are do-loops. For the greater part of this chapter we will only consider innermost or stand-alone do-loops, but a brief discussion of loops with embedded loops appears in section 16.11. Specific characteristics of a given do-loop determines its vectorizability, and when the V-option is specified the compiler will either succeed or fail in its vectorization attempt. Part of the FORTRAN generated output listing will consist of a detailed account of the number of appearing loops in each subroutine, and also of the number of vectorizable (collapsable) loops, in conjunction with reasons why the others did not qualify.

The complete set of criteria for vectorizable do-loops appears in the FORTRAN manual, pages 11.1-4, and a summary can also be found in section 1Ø of this chapter. Guided by that material you may now, with the output listing in hand, want to improve program performance by tinkering with the loops that did not vectorize.

Although such an effort may "save" a few loops, chances are that some of them just aren't possible to restructure when considered as separate pieces of code. The basic problem is that many codes are actually designed to be incompatible with vector processing. That is something you may have to live with if you are dealing with an already written scalar code, but which otherwise can be avoided by "thinking in vector mode" already in the design phase of a problem solution.

The coding phase is the second best place to implement "vector thinking". In many cases a do-loop, or a set of nested such, can be written in several different ways, all of which at run time will produce the same results. The traditional approach when the code is aimed for a scalar machine is, at best, that the programmer chooses a way that minimizes the number of arithmetic operations. The reasoning behind that is of course that no other factor determines the actual execution speed. As has previously been pointed out, however, that is not even true for all scalar machines - in particular not for those that feature virtual memory. And it's definitely not true on machines capable of vector processing - regardless of their sophistication. Several other factors, like array storage, sequential access, recursiveness, etc., must be understood and paid attention to in order to make the code vectorizable to the maximum extent.

The remainder of this chapter is designed to provide you with a basic understanding of the vector concept, as applied to the CYBER 203/205. More explicitly, it will try to teach you how the automatic vectorizer "thinks", so that you can take full advantage of its power. That will, hopefully, improve your ability to make existing scalar code vectorizable (if necessary), but also enable you to write good, transportable, vectorizable code from scratch.

16.3 DIFFERENT TYPES OF VECTOR INSTRUCTIONS

With the exception of linked triads, treated in the next section, a general vector instruction can be expressed in the following form:

$$VR = V1(OP)V2$$

VR is the result vector, V1 and V2 are two input vectors and (OP) stands for an arithmetic (+,-,*,/,**) or logical (.AND., .OR., .XOR.) operator. One of the two input operands may instead be a

scalar, in which case we are talking about a vector operation with "broadcast". The scalar value is namely broadcast to simulate a vector with the same length as the other input operand. The following forms also exist:

```
VR = V1     VR = - V1     VR = .NOT. V1
```

Each of the following scalar do-loops will be compiled as a single vector instruction if the V-option is specified:

```
   DO 10 J = 1,N
10 A(J) = B(J) - C(J)          (Vector Subtract)

   DO 20 J = 1,N               (Vector multiply
20 X(J,4) = 3.0*Y(J)           with broadcast)

   DO 30 J = M,N               (Vector logical AND.  R and S
30 R(J-1) = S(J-1).AND.S(J)    must be of type logical)
```

Certain scalar function references correspond directly to another type of vector instructions, namely the ones that only take one input vector. As of today (FORTRAN, Release 1.5.1) the scalar set consists of:

```
ABS  IABS  FLOAT  IFIX  SQRT
```

but an expansion is quite possible in future releases. The following do-loops will each be compiled as a single vector instruction, assuming that the V-option is specified:

```
      DO 10 J = L,N
      A(J) = SQRT (B(J,K))
10 CONTINUE

      DO 20 J = 1,M
      A(J) = FLOAT(KA(J))
20 CONTINUE

      DO 30 J = 1,M
30 A(J) = KA(J)
```

Some of the more commonly used SYSLIB functions, namely:

SIN COS TAN ASIN ACOS ATAN EXP ALOG ALOG1Ø

have been coded as special subroutines accepting a vector argument and delivering a vector result. The compiler knows that, and the above mentioned function references are thus "vectorizable" as a kind of pseudo vector instructions. In reality they are of course just subroutine calls. The following do-loops are all "vectorizable":

```
      DO 1Ø J = 1,N
      A(J) = SIN(B(J))
1Ø    CONTINUE

      DO 2Ø J = 1,N
      X(J) = ALOG(Y(J,K))
2Ø    CONTINUE

      DO 3Ø J = M,N
      T(J) = ACOS(T(J))
3Ø    CONTINUE
```

With respect to the function references mentioned in the last paragraph, the difference in execution speed between keeping the code in scalar mode and allowing it to vectorize is quite substantial. Assuming an iteration count of N, large enough to allow us to ignore startup times, the following approximate timings were obtained for the CYBER 2Ø3 [unit = 2Ø ns cycle]:

Scalar loop containing	Timing in scalar mode	Timing when vectorized
SIN/COS	23ØN	52N
TAN	31ØN	79N
ASIN/ACOS	317N	113N
ATAN	281N	99N
EXP	182N	51N
ALOG/ALOG1Ø	273N	98N

16.4 THE LINKED TRIAD

The general form of a linked triad is

SR = S1(OP1)S2(OP2)S3

S1, S2, and S3 are input operands, SR an output operand (result), and OP1 and OP2 symbolize some type of operators. Some FORTRAN examples, using scalar operands are

```
X = Y+Z+T        X = Y+Z*T
X = Y*Z/T        R = S.AND.F.OR.G
```

The CYBER 2Ø5 has the capacity of computing certain forms of linked triads in vector mode as if they each represented a single vector instruction. These forms are defined as follows:

1) At least one, but no more than two of the input operands are vectors. That implies that the output operand also must be a vector.

2) One of the two operators is a floating point multiply, and the other is a floating point add or subtract.

Using V for vector and S for scalar, we thus have the following forms:

```
VR = S1 + V1*V2        VR = S1 - V1*V2
VR = S1 + S2*V2        VR = S1 - S2*V2
VR = V1 + S1*V2        VR = V1 - S1*V2
VR = V1 + S1*S2        VR = V1 - S1*S2
```

The computation in just about any linear algebra routine is dominated by operations of the forms given in the third line above, and only rarely will you encounter the others. The vector processor's capacity for linked triads actually extends somewhat beyond the limits given here, but to take advantage of that you have to deviate from conventional FORTRAN syntax - an action which the title of this chapter prohibits. The timing for any of the above linked triads on the CYBER 2Ø5 (2 pipe) is:

84 + N/2 cycles

That is the same stream rate as that of all other floating point vector instructions (excluding the divide), and yet twice as many arithmetic operations are performed! It should be stressed that the CYBER 2Ø3 does not have linked triad capability. Since mathematicians traditionally think "rowwise" rather than

"columnwise", many FORTRAN coded algorithms that display linked triads are not directly vectorizable. In such cases, however, a reordering of the sequence of arithmetic operations is often all that is needed to obtain the desired vector structure. Consider, as an example, the problem of multiplying a rectangular matrix A with a column matrix X, obtaining a column matrix B. In matrix notation that corresponds to performing the operation

$$B = AX$$

The elements of B are formed as inner products (dot products) as expressed by the following formula, where we have assumed that A has dimensions (M,N):

$$b_j = \sum_{k=1}^{n} (a_{jk}\, x_k) \qquad (j = 1,2,\ldots,m)$$

The conventional approach to coding this algorithm in FORTRAN is to reproduce the mathematical formula as closely as possible:

```
      DO 1Ø J = 1,M
      B(J) = Ø.
      DO 1Ø K = 1,N
1Ø    B(J) = B(J) + A(J,K)*X(K)
```

That piece of code will, of course, execute properly, but it doesn't exhibit much of a vector structure. In the innermost loop only X is accessed sequentially, while A is accessed rowwise and B effectively represents a scalar. Thus, on the CYBER 2Ø3 the automatic vectorizer can accomplish nothing, while on the CYBER 2Ø5 it may be smart enough to do a GATHER on the A-elements and then use Q8SDOT for the dot-product. The latter solution is not very efficient, though, partly because thrashing (mortal paging) may occur if the size of A exceeds available memory. Note that although the innermost loop in each pass computes a scalar linked triad, a translation to a vector linked triad is not possible. To obtain vector structure we must effectively exchange the outer and inner loops:

```
      DO 20 J = 1,M
20    B(J) = 0.

      DO 30 K = 1,N
      DO 30 J = 1,M
30    B(J) = B(J) + A(J,K)*X(K)
```

We now have two loops with explicit vector structure: The 20-loop and the innermost 30-loop. The former will vectorize as one vector instruction on both machines: a vector assignment statement with broadcast (same speed as a vector add). To see that the latter also is vectorizable, we perform a complete factorization, which requires the introduction of a temporary array [the concept of factorization will be discussed in detail in the next section]:

```
      DO 70 K = 1,N
      DO 50 J = 1,M
50    T(J) = A(J,K) * X(K)
      DO   60   J = 1,M
60    B(J) = B(J) + T(J)
70    CONTINUE
```

There is no longer any doubt that the 30-loop is vectorizable. On the CYBER 203 it will translate into a vector multiply, writing the result vector into temporary space in the dynamic stack, and a vector add of that result to (B(J), J=1,M). On the CYBER 205 the compiler will indeed make use of the vector linked triad instruction: the form is "VR = V1 + V2*S1", where X(K) corresponds to the scalar. It may be of interest to compare the different timings of the entire matrix multiplication B = AX. For simplicity we will assume that M and N are large, so that we can disregard startup times. The compile card presumably contains "O=BOV", and the unit used is as usual the cycle time, 20 ns:

	CYBER 203	CYBER 205 (2 pipe)
Naive (10-loops)	17MN	2.25MN
Fancy (20-30-loops)	3MN	0.50MN

16.5 FACTORIZATION OF DO-LOOPS

A "one-liner" is defined as a do-loop with exactly one FORTRAN statement, not counting the DO and CONTINUE statements. If a particular do-loop can be broken down into a sequence of one-liners, we say that the loop is "factorizable". An example would be:

```
      DO 1Ø J = 1,N
      XX(J) = X(J)**2
      YY(J) = Y(J)**2
      SN(J) = SQRT(XX(J)+YY(J))
1Ø CONTINUE
```

which could be factorized as:

```
      DO 11 J = 1,N
11 XX(J) = X(J)**2

      DO 12 J = 1,N
12 YY(J) = Y(J)**2

      DO 13 J = 1,N
      SN(J) = SQRT(XX(J)+YY(J))
13 CONTINUE
```

A "complete factorization" would result if each one-liner featured only one arithmetic operator (+,-,*,/,**) or logical operator (.AND., .OR., .XOR., .NOT.) or function reference. To make the factorization above complete we would have to break down the 13-loop further, which in turn would require the introduction of a temporary array, say XY:

```
      DO 14 J = 1,N
14 XY(J) = XX(J)+YY(J)

      DO 15 J = 1,N
      SN(J) = SQRT(XY(J))
15 CONTINUE
```

We could of course have used SN as the temporary array, but that type of solution is not always possible. When a scalar appears on the left hand side of an equal sign, the introduction of a temporary array is often unavoidable, as in the following case:

ORIGINAL

```
      DO 10 J = 1,N
      S1 = A(J,K+1) - A(J,K-1)
      S2 = B(J,K+1) - B(J,K-1)
      X(J) = ABS(S1) + ABS(S2)
10    CONTINUE
```

FACTORIZED

```
      DO 11 J = 1,N
11    ST1(J) = A(J,K+1) - A(J,K-1)

      DO 12 J = 1,N
12    ST2(J) = B(J,K+1) - B(J,K-1)

      DO 13 J = 1,N
      X(J) = ABS(ST1(J)) + ABS(ST2(J))
13    CONTINUE
```

Complete factorization would differ only with respect to the 13-loop, which would split into:

```
      DO 14 J = 1,N
      ST1(J) = ABS(ST1(J))
14    CONTINUE

      DO 15 J = 1,N
      ST2(J) = ABS(ST2(J))
15    CONTINUE

      DO 16 J = 1,N
16    X(J) = ST1(J) + ST2(J)
```

The point of this exercise is to synthesize it with the previous section, and make the following statement:

> A do-loop that is not completely factorizable is not vectorizable.

The value of using this factorizability test as a first check for vectorizability lies in the fact that you thereby are forced to "see" the vector structure, or lack of such, in a particular do-loop. The one-liner resulting from a complete factorization are namely exactly the entities that the automatic vectorizer on a one-for-one basis would compile as vector instructions - if possible. The sometimes necessary vector temporaries are indeed created by the compiler - the dynamic stack comes in handy for that purpose.

We can now state the criterion for vectorizability in the following simple form:

> A do-loop that is completely factorizable
> is vectorizable if, and only if, all of the
> resulting one-liners are vectorizable.

Each one-liner must therefore not feature more than one operator (arithmetic or logical) or one function reference, where the function is one from the set given in the previous section. Furthermore, the present compiler restricts the data types to real, integer and logical. Other requirements, such as contiguity of array references and lack of recursiveness are discussed in the remainder of this chapter, and a summary of the rules is given in the section 16.10.

16.6 CONTIGUITY IN MEMORY

Recall the definition of a vector:

Vector = a set of contiguous storage locations in memory.

The implication of this definition is that a do-loop that accesses arrays in a nonsequential manner lacks vector structure, and therefore is not directly vectorizable. The automatic vectorizer on the CYBER 203 will thus reject the following loop:

```
      DO 10 J = 1,N,2
10    A(J) = B(J) + C(J)
```

On the CYBER 205, however, the compiler does not give up that easily. Two machine instructions, commonly referred to as GATHER and SCATTER, are namely available for the purpose of moving data elements from nonsequential to sequential locations (GATHER) or

the reverse (SCATTER). The only type of nonsequential locations that the compiler can deal with in this manner are the ones where consecutive locations are separated by a constant stride (=number of words in memory). In the example above the stride is 2, since consecutive passes through the loop adds B(1)+C(1), B(3)+C(3), B(5)+C(5) etc., and B(1), B(3), B(5) can be expressed as B(1), B(1+2), B(1+2+2). A stride of 1 simply means that the elements are contiguous. The automatic vectorizer on the CYBER 205 will thus vectorize the loop above as follows:

1) Gather (B(J),J=1,N,2) into the first N/2 locations of the dynamic stack. We will refer to that area as VB.

2) Gather (C(J),J=1,N,2) into the next N/2 locations of the dynamic stack (VC).

3) Perform the vector addition VB+VC, and store the result in the next N/2 locations of the dynamic stack (VA).

4) Scatter VA into (A(J),J=1,N,2).

The timing of the loop when run in scalar mode, and when automatically vectorized, is on the CYBER 205 [L=N/2]:

Scalar mode	17 + 19L
GATHER	39 + 5L/4
GATHER	39 + 5L/4
ADD	51 + L/2
SCATTER	71 + 5L/4
Vector mode	200 + 4.25L

By equating the timings for scalar and vector mode we obtain L=12.4, which tells us that the vectorized version is faster when L is 13 or greater. That is, of course, quite satisfactory, but when GATHER/SCATTER are involved the break even point is often quite a bit higher. And by comparing the timings of the add instruction itself to that of the total operation we see that for large L-values another factor of 8 could have been gained, had the data been contiguous in memory to start with.

The conclusion must be that although it's nice to know that the compiler can "vectorize" some code lacking explicit vector structure, it's not something we would want to take advantage of unless it's absolutely necessary. By the way, the GATHER/SCATTER instructions do indeed exist on the CYBER 203 as well. On that

model, however, they are prohibitively slow (about 40 cycles stream rate) and the automatic vectorizer therefore does not use tnem.

A simple exchange of loop indices is sometimes all that is needed to introduce the desired vector structure:

```
      DIMENSION A(80,120),X(80,120)
      DO 10 J = 1,M
      DO 10 K = 1,N
10    A(J,K) = ABS(X(J,K))
```

The innermost loop will not vectorize on the CYBER 203, while on the CYBER 205 the GATHER/SCATTER instructions (stride=80) will enable the compiler to perform the ABS in vector mode. However, the superior version is the following:

```
      DO 10 K = 1,N
      DO 10 J = 1,M
10    A(J,K) = ABS(X(J,K)
```

Now the innermost loop with vectorize on both machines - and no GATHER/SCATTER are needed. The speedup is substantial.

Often it is necessary to alter the way in which arrays are stored in order to obtain vector structure. Consider as an example the following situation:

```
      DIMENSION X(2,200),A(200),B(200)
      DO 10 J = 1,200
      X(1,J) = X(1,J) + A(J)
10    X(2,J) = X(2,J) + B(J)
```

Recalling that two-dimensional arrays are stored by the columns it's evident that the references to X are nonsequential in nature. The loop will still vectorize on the CYBER 205, but not in the most efficient manner. One way to achieve explicit vector structure is to change the columnwise storage mode to rowwise, something that can be accomplished by a type declaration statement:

```
      ROWWISE X(2,200)
      DIMENSION A(200,B(200)
      DO 10 J = 1,200
      X(1,J) = X(1,J) + A(J)
10    X(2,J) = X(2,J) + B(J)
```

The loop is now completely vectorizable on both machines, since X(1,1), X(1,2), X(1,3), etc., are adjacent in memory. The drawback is that it is easy to forget that one, or a few, arrays are stored in a non-default mode. Difficulties may also arise when your subroutine is used by others who, no doubt, will assume that all storage is columnwise. A more attractive method is therefore to just swap dimensions in the DIMENSION statement:

```
      DIMENSION X(200,2),A(200),B(200)
      DO 10 J = 1,200
      X(J,1) = X(J,1) + A(J)
10    X(J,2) = X(J,2) + B(J)
```

Both solutions accomplish the same thing on the machine level, but the latter is more transparent to the user.

16.7 MAXIMUM VECTOR LENGTH

Another concern should be the maximum iteration count. The maximum allowed vector length is namely 65535, and unless the compiler with a reasonable certainty can establish that that length will not be exceeded, no vectorization will take place. "DO 10 J=1,60000" is thus OK, while "DO 10 J=1,70000" is not. But what about "DO 10 J=1,N"? Well, in this case the compiler will check the dimensions of the arrays appearing in the loop, and if they are within limits the loop is considered vectorizable. Hence,

```
      DIMENSION A(50000,4),B(50000)
      DO 10 J = 1,N
10    A(J,2) = B(J)**2
```

is vectorizable, regardless of the value of N, since the "50000" in the DIMENSION statement is what forms the basis for the compiler decision - N is not even known at compile time. If "50000" is replaced with "70000", the loop will not vectorize.

A problem occurs when one or several of the arrays in the loop are dummy arrays at the same time as the loop count is unknown, as is the case in the following loop:

```
      SUBROUTINE SUB(A,B,M,K)
      DIMENSION A(M,1),B(M)
      DO 10 J = 1,M
10    A(J,K) = SQRT(B(J))
```

Here the compiler can not check for maximum vector length, and must thus refuse to vectorize. By specifying both V and U (U for Unsafe) on the FORTRAN card, however, you tell the compiler that you guarantee that no loop counts will ever exceed 65535. Relieved of the responsibility, the compiler will then vectorize loops like the one above.

16.8 RECURSION

A recursive do-loop is defined as a loop in which the result of the work performed in the Kth pass depends on the result of one or several previous passes, typically the (K-1)st. An example would be:

```
      DO 10 J=2,N
10    L(J) = L(J) + L(J-1)
```

If the elements of array L were initialized to 1, then the execution of this scalar loop would look as follows:

```
L(2) = L(2) + L(1)   = 1 + 1     = 2
L(3) = L(3) + L(2)   = 1 + 2     = 3
 .    .    .
L(N) = L(N) + L(N-1) = 1 + (N-1) = N
```

The result would thus be (L(J)=J, J=1,N). If an attempt were to be made to perform the same work in vector mode, the result would be completely different - probably L(1)=1 and (L(J)=2, J=2,N). The reason for that is that the two input vectors have to start streaming from memory to the vector pipes before any results are calculated, implying that the only operands available to perform additions with are "old" L-values. To clarify that, let's use

the model introduced in section 15.3, and take a look at the situation at cycle 3:

```
Input path          SEG1      .  .  .

(L3,L4)   (L2,L3)   (L1,L2)
```

The first pair of operands has here entered segment 1, but it will take an additional 6 cycles before the result, which will be the "new" L(2), pops out from the end of the pipe. By that time the second pair of input operands have almost finished its path through the pipe. It's clearly impossible that the "new" L(2) could be a member of that pair, since that has just barely been computed, and now is on its way back to memory. The situation at cycle 9 can be depicted as follows:

```
Input path          SEG1      .  .  .  SEG6     Output path

(L10,L9) (L9,L8)   (L8,L7)   .  .  .  (L3,L2)  L2
(old)    (old)     (old)     .  .  .  (old)    (new)
```

The conclusion must be that recursive loops, such as the one discussed, cannot be vectorized. The compiler knows that, and does not fool around with such constructs. Note that the following loop is not recursive:

```
      DO 10 J=2,N
10    L(J-1) = L(J-1) + L(J)
```

This is an example of forward reference, and presents no difficulties. On the other hand, the following loop will not vectorize:

```
      KP1 = K+1
      DO 100 J=KP1,N
10    L(J-K) = L(J-K) + L(J)
```

If K is positive, this loop is of course not recursive. However, the value of K is not known at compile time, and the compiler doesn't take any chances - it would be deadly if K turned out to be negative.

The clever reader might here interject that if the length of the input and output paths were known, as well as the number of segments, then one could predict how large an offset was necessary to assure that "new" values rather than "old" were obtained. If the sum of the lengths were 5Ø, for instance, then the following loop should vectorize:

```
      DO 1Ø J=52,N
1Ø L(J) = L(J) + L(J-51)
```

In principle that is of course true. In practice, however, the critical length varies with the relative storage locations of the arrays appearing in the loop, and also with the architecture of the machine, to just name a few parameters. So the rule is that potential recursion is treated as absolute recursion by the compiler, and when you later learn how to perform explicit vectorization you are strongly urged to take the same position.

At this point another warning might be appropriate: do not stake your life on that the loop in the beginning of this section ("L(J) = L(J)+L(J-1)") produces (L(J)=2, J=2,N). If, namely, a system interrupt (e.g. a page fault) occurs after, say, the 3Ø first results are computed, then you will get L(1)=1, (L(J)=2, J=2,31),L(32)=3, (L(J)=2, J=33,N). Right?

16.9 STACKLIB

The automatic vectorizer has a certain, limited, capacity for dealing with recursive loops. That capacity may later be expanded, but as of today only the following types can be handled:

```
1. X(J) = X(J-1) + Y(J)
2. S    = S + X(J)
3. S    = S + X(J)*Y(J)
4. S    = S + X(J)*X(J)
5. S    = S + X(J)**2
```

Each example is to be perceived as a do-loop of type "one-liner" with loop-increment 1, where the order of the operands is immaterial. Thus, the first example represents any one of the following four loops [X and Y must be distinct]:

```
      DO 10 J = L,M                    DO 10 J = L,M
   10 X(J) = X(J-1) + Y(J)          10 X(J) = Y(J) + X(J-1)

      DO 10 J = L,M                    DO 10 J = L,M
      X(J) = X(J-1) + Y(J)             X(J) = Y(J) + X(J-1)
   10 CONTINUE                      10 CONTINUE
```

On the CYBER 203, all 5 categories will be transformed into calls to highly optimized scalar subroutines, called STACKLIB routines. The tasks are there performed in unrolled scalar loops, which take advantage of the large register file. On the CYBER 205, with the greatly enhanced vector processor, there are other options available for all but the first category. In addition to the "normal" vector instructions, namely, a number of more complicated vector-like operations can be performed in the pipes. Such tasks are initiated by a certain type of machine instructions called Vector Macros. Two of these were mentioned earlier in this chapter: GATHER-SCATTER, or Q8VGATHR-Q8VSCATR, which are their FORTRAN callable names. Two others are Q8SSUM and Q8SDOT, which compute the sum of the elements of a vector and the dot product of two (distinct or identical) vectors, respectively. Although also the CYBER 203 feature these macros, the timing specification below makes it obvious why the compiler prefers unrolled scalar loops there.

	CYBER 203	CYBER 205 (2 and 4 pipe)	
Q8SSUM	260 + 11N	96 + N	(cycles)
Q8SDOT	350 + 12N	107 + N	(cycles)

Thus, on the CYBER 205 loops of category 2 will be translated with the machine instruction Q8SSUM, while Q8SDOT will be used for those of categories 3, 4, and 5 - assuming, of course, that the V-option is specified.

Note that the categories 2-5 indeed represent recursive operations: the partial sum in step J can not be computed without knowledge of the partial sum computed in step J-1.

It should be mentioned that most of the Vector Macros are directly accessible to you in the form of FORTRAN-like function references, as described in Chapter 14 of the FORTRAN manual, and discussed in a later chapter of this document.

16.10 CRITERIA FOR VECTORIZABILITY OF INNERMOST DO-LOOPS

Below follows a list of the criteria that determine the vectorizability of an innermost do-loop. All conditions must be satisfied, but it should be emphasized that the set reflects the status of the FORTRAN compiler with release number 1.5.1 and 1.5.2. Future releases can be expected to relax some of the conditions, in particular those numbered 1, 3, and 7.

The list is intended as a guide, and as a summary of the presentation given in the previous sections of this chapter. Conditions 4-7 deserve some extra attention since they are not discussed there. The question of vectorizability of second innermost do-loops has been ignored so far, but will be briefly discussed on in the next section.

1) All data elements must be of type real, integer or logical. More than one type may occur.

2) All appearing operators must belong to the following set:
   + - * / ** .AND. .OR. .XOR. .NOT.
   In particular, relational operators are not permitted.

3) No function or subroutine references may occur, except to the following functions:

   ABS IABS FLOAT IFIX SQRT
   SIN COS TAN ASIN ACOS ATAN
   EXP ALOG ALOG10

4) Variables on the left hand side of an equal sign may not appear in EQUIVALENCE statements. (No longer a condition in FORTRAN, Release 1.5.2, provided the U-option is selected).

5) No IF or GOTO statements may appear.
   (Otherwise factorization would not be possible).

6) No explicit vector statements may appear.
   (Explicit vectorization is treated in Chapters 17-18).

7) Loop-dependent array subscripts must be expressed in the form J, J+N, or J-N, where J is the loop-index and N is an integer constant (not a variable!). On the CYBER 205 the form J*N is also permitted.

16.0 AUTOMATIC VECTORIZATION
16.10 CRITERIA FOR VECTORIZABILITY OF INNERMOST DO-LOOPS

8) Array references must correspond to sequential memory access. This condition is somewhat relaxed on the CYBER 205, as discussed in section 16.6.

9) The loop must not be recursive (section 16.8). Exceptions are constructs that can be stacklibed (section 16.9).

10) The loop must be completely factorizable (section 16.5). This is automatically true if conditions (5) and (9) are satisfied.

11) The iteration count must not exceed 65535, as discussed in section 16.7.

16.11 VECTORIZATION OF SECOND INNERMOST DO-LOOPS

A do-loop containing a vectorizable do-loop is under certain circumstances vectorizable in the sense that the vector lengths established by the innermost loop can be extended, as in the following example:

```
      DIMENSION A(200,10),B(200,10),C(200,10)
      DO 10 K = 1,5
      DO 10 J = 1,200
   10 A(J,K) = B(J,K) + C(J,K)
```

The innermost loop is clearly vectorizable, with a vector length of 200. Since the DIMENSION statement declares column lengths of exactly 200 for all the arrays involved, it is now possible to perceive the doubly nested loop as one long vector addition of the first 5*200=1000 elements of arrays B and C. This is based on the knowledge of how the array elements are stored in memory: (B(J,1), J=1,200) is immediately followed by (B(J,2), J=1,200), etc.. If, however, any one of the arrays A, B, or C had been dimensioned with a first dimension different from the iteration count in the innermost loop, the automatic vectorizer would not have been able to touch the outermost loop. The following example illustrates that:

```
      DIMENSION A(200,10),B(200,10),C(200,10)
      DO 10 K = 1,5
      DO 10 J = 1,199
   10 A(J,K) = B(J,K) + C(J,K)
```

16.0 AUTOMATIC VECTORIZATION
16.11 VECTORIZATION OF SECOND INNERMOST DO-LOOPS

The active elements in this doubly nested loop are the first 199 elements of the first 5 columns of each array, or locations number 1-199, 201-399, 401-599, 601-799, 801-999. Clearly this does not represent contiguous locations in memory, and vectorization beyond the innermost level of loops is not to be expected.

A variable initial, terminal, or incrementation parameter in the innermost loop prohibits the vectorization of any containing loop. In such cases, namely, the compiler has no way of determining whether the innermost loop really operates on full columns. If, on the other hand, the outer loop has a variable loop parameter, vectorization can take place provided the question about maximum vector length (section 16.7) presents no hinder. The following loop is therefore "doubly" vectorizable if the arrays are dimensioned (200,10), but not if they are dimensioned (200,330):

```
      DO 10 K = 1,N
      DO 10 J = 1,200
10    A(J,K) = B(J,K) + C(J,K)
```

When the vectorization of the innermost loop requires GATHER/SCATTER utilization (CYBER 205 only), the problem with contiguity becomes too ditficult for the compiler, and the outermost loop in the following example is therefore not vectorizable (yet):

```
      DO 10 J = 1,200
      DO 10 K = 1,10
10    A(J,K) = B(J,K) + C(J,K)
```

If, in a doubly nested loop, any statement appears between the two DO statements, the outermost loop is not vectorizable. This is fundamentally true, and does not reflect an incompetence of the compiler. Consider, namely, the following code:

```
      DIMENSION A(50,80),X(80),Y(80),R(50),S(50)
      DO 10 J = 1,80
      T = X(J)**2 + Y(J)**2
      DO 10 K = 1,50
10    A(K,J) = T*R(K) + S(K)/T
```

16.0 AUTOMATIC VECTORIZATION
16.11 VECTORIZATION OF SECOND INNERMOST DO-LOOPS

To display clearly the vector structure of this construct we must introduce a temporary array, F(80), and then split the loops:

```
      DO 20 J = 1,80
20    F(J) = X(J)**2 + Y(J)**2


      DO 21 J = 1,80
      DO 21 K = 1,50
21    A(K,J) = F(J)*R(K) + S(K)/F(J)
```

To split loops into separate blocks of code like this is not possible for the compiler. But for the programmer it's of course an easy task, and the result is that the computations of the squares and their addition now will be done in vector mode, since the 20-loop easily vectorizes. A closer look at the doubly nested 21-loop, however, reveals that there is still no easy way to handle these computations in vector mode if the vector length is required to be 50*80=4000.

16.12 AUTOMATIC VECTORIZATION - IS IT SUFFICIENT?

A programmer that well understands the vector concept and its implementation can, no doubt, write good, transportable, scalar code in such a way that use of the automatic vectorizer will greatly speed up the execution on the CYBER 203/205. That's an indisputable fact, and it goes without saying that the same is true on other vector processors with good automatic vectorizers. Nevertheless, arguments can be made for going a step further:

1) The presence of a startup time in the general formula for a vector instruction (time=startup+N*stream rate) indicates that loops with small N-values may be more efficiently executed in scalar mode. But the automatic vectorizer cannot be enticed to avoid the vectorization of such loops - it's all or nothing. Even if an "exclusion mechanism" did exist, it would probably not be very effective since most iteration counts are not known at compile time. (Subroutines can, of course, be compiled separately or in blocks, a method that partially solves the problem).

16.0 AUTOMATIC VECTORIZATION
16.12 AUTOMATIC VECTORIZATION - IS IT SUFFICIENT?

2) By totally relying on the automatic vectorizer the user is easily deceived into believing that he/she automatically obtains the maximum performance that the computer can offer. Such a conclusion is, of course, completely false - regardless of the advertised sophistication of the computer in question. Not only can the programmer "create" vector structure by manually reorganizing the data flow (primarily), as discussed in the previous sections of this chapter, but limitations of the FORTRAN language simply prevents any compiler from utilizing the full capacity of today's super computers. In addition, FORTRAN is not complex (rich) enough to allow the user to furnish the compiler with all available information, resulting in the rejection of some otherwise vectorizable constructs.

From the above, it is clear that any vector processor claiming to offer the state-of-the-art of vector computing must, in addition to an automatic vectorizer, give the user convenient access to those of its capacities that are not available through standard FORTRAN. The CYBER 200 series of computers meets these demands by featuring a FORTRAN language containing vector extensions, giving the user complete access to its computing power. These vector extensions exist superimposed on the normal, scalar language syntax, and many are very "FORTRAN-like", making them convenient to use. The manual introduction of vector syntax into your code is termed "explicit vectorization", and is the subject of the next chapter .

17.0 EXPLICIT VECTORIZATION

17.1 INTRODUCTION

A simple fact of life is that sometimes you just won't be satisfied with what the automatic vectorizer produces. Maybe your do-loops predominantly have very small iteration counts, so that instead of having them all vectorized you would be much happier if only a selected few were run in vector mode. Maybe you have do-loops that to your trained eye clearly exhibit vector structure, but which don't qualify for automatic vectorization. Or maybe you simply belong to the school of programmers that likes to exercise as much power as possible over the machine.

Regardless of the reason for your interest, the answer to your needs is provided by the CYBER 200 FORTRAN compiler. Its main body functions much the same way as other FORTRAN compilers, accepting scalar FORTRAN code as input and producing scalar machine code as output. But in two ways it is greatly enhanced:

1) A vector syntax is provided as an extension of the standard scalar syntax, allowing the programmer to explicitly vectorize code structures at his/her own discretion. The vector syntax itself is in turn extended by a large number of highly optimized SYSLIB functions (trigonometric and others) that accept vector arguments and deliver vector results. All such functions have convenient FORTRAN-like calling sequences.

2) The complete set of machine instructions are available to the FORTRAN programmer in the form of a special call-syntax. That may well be more then you asked for, and if you don't do particularly tricky manipulations you will probably never need to make use of it. But it's nice to know that the power is there - in case of emergency.

17.2 VECTOR SYNTAX - THE EXPLICIT TYPE

To completely define a vector in CYBER 200 FORTRAN, three things must be specified: data type, starting address, and length. A vector is defined as contiguous storage locations in memory, and since arrays are also confined to memory, the starting address is conveniently represented by an array element. Eacn array in your code has a well defined data type, declared either implicitly or by use of type declaration statements, and the array element used as a pointer will therefore also automatically define the data type of the vector elements. The length, finally, is specified as an additional subscript preceded by a semicolon (;). Some examples to illustrate this:

```
DIMENSION A(100),KQ(50,5)
COMPLEX C(100,100)
ROWWISE R(1000,2000)
```

`A(1;60)`	A real vector consisting of (A(J),J=1,60)
`A(5;90)`	A real vector consisting of (A(J),J=5,94)
`KQ(1,2;100)`	An integer vector consisting of ((KQ(J,K),J=1,50),K=2,3)
`C(1,1;5*100)`	A complex vector consisting of ((C(J,K),J=1,100),K=1,5) This represents 1000 words of memory.
`R(5,5;996)`	A real vector consisting of (R(5,J),J=5,1000)

Legal data types are REAL, INTEGER, COMPLEX, DOUBLE PRECISION, and BIT. All subscripts, including the length, may be specified with integer expressions rather than integer constants.

We now have the freedom to manually vectorize any loop displaying vector structure, regardless of whether the automatic vectorizer would have been happy with it or not. In particular, the data types COMPLEX and DOUBLE PRECISION are no longer excluded, loop-dependent subscripts are allowed to be quite a bit more complicated, and EQUIVALENCE statements present no hinder. The following scalar loops, followed by their vector equivalents, illustrate the simple implementation of the vector syntax:

17.0 EXPLICIT VECTORIZATION
17.2 VECTOR SYNTAX - THE EXPLICIT TYPE

Scalar(10)

```
      DO 10 J = 1,100
10    R(J) = S(J) + T(J+KX)
```

Vector(10)

```
      R(1;100) = S(1;100) + T(1+KX;100)
```

Scalar(20)

```
      DIMENSION A(200,10),Y(200,10),Z(200,10)
      DO 20 K = 1,N
      DO 20 J = 1,200
20    A(J,K) = Y(J,K) + Z(J,K)
```

Vector(20a)

```
      DO 20 K = 1,N
20    A(1,K;200) = Y(1,K;200) + Z(1,K;200)
```

Vector(20b)

```
      A(1,1;200*N) = Y(1,1;200*N) + Z(1,1;200*N)
```

Scalar(30)

```
      COMPLEX CX(100),CY(100)
      DO 30 J = L,M
30    CX(J) = CY(J) *  CON
```

Vector(30)

```
      CX(L;M-L+1) = CY(L;M-L+1)* CON
```

Scalar(40)

```
      DO 40 K = 1,N
      DO 40 J = 1,M
40    B(J) = B(J) + A(J,K)*X(K)
```

Vector(40)

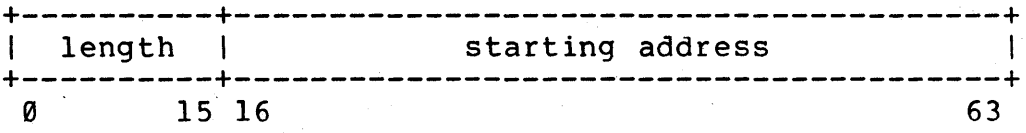
```
      DO 40 K = 1,N
40    B(1;M) = B(1;M) + A(1,K;M)*X(K)
```

With the exception of Vector(40), each vector statement above will be compiled as a single vector instruction, just as if the automatic vectorizer had vectorized the corresponding scalar loop. In the Vector(40) case, however, that is true only on the CYBER 205, where the linked triad instruction will be used. On the CYBER 203, the vector expression will be split by the compiler into two instructions: one vector multiply with broadcast, placing the result vector in the dynamic stack, followed by a vector add.

Since the semicolon notation explicitly displays all relevant characteristics of a given vector, it is often referred to as the explicit vector notation. The most important other type, which is the implicit, or descriptor notation, is discussed in the next section.

17.3 VECTOR SYNTAX - THE IMPLICIT TYPE (DESCRIPTORS)

On the machine level a vector is represented as a 64-bit word containing the integer length in the leftmost 16 bits, the length field, and the address of the starting location in the rightmost 48, the address field:



Such a word is called a descriptor, and may be considered as a scalar variable invented by the compiler. In general, the register file is used to hold its value. If array A starts at virtual address #400000, the vectors A(1;64) and A(2;64) are represented as follows:

```
A(1;64)     0040 000000400000
A(2;64)     0040 000000400040
```

The information about data type is not present in the descriptor, but the compiler knows, and will implicitly forward its knowledge to the vector processor by choosing the right type of machine instruction. It is instructive to take a look on the assembly language (META) representation of the machine code that the compiler generates for a floating point vector add:

```
FORTRAN   A(1;N) = B(1;N) + C(1;N)

META      PACK   N,ADDA,AD
          PACK   N,ADDB,BD
          PACK   N,ADDC,CD
          ADDNV  BD,CD,AD
```

All symbols represent registers. The PACK instruction strips off the rightmost 16 bits from the register corresponding to the first (integer) argument, N, and places them in the length field of the third, AD. The second register is assumed to contain an address in the rightmost 48 bits, which is copied into the same field of the third. The so assembled descriptors are then used as arguments of the floating point vector add instruction, ADDNV.

Thus, each vector instruction requires a certain amount of overhead in the form of scalar instructions. That overhead, however, usually executes in "zero" time, since while one vector instruction is running, the scalar processor can work to prepare for the next. Several vector instructions on the CYBER 203, and almost all on the CYBER 205, namely, permit overlapping scalar processing provided no memory references (LOAD/STORE) appear.

One very important piece of information can be gathered from the META code above: no checking of the value of N is performed at execution time. The machine couldn't care less about how many bits are used to represent N in the register - the rightmost 16 bits are just stripped off and regarded as the vector length. The largest number representable by 16 bits is 2**16-1=65535, which explains why no vector can be longer. But if the value of N happened to be greater, say 66000, the length actually used would be MOD(65536,N), or 464. Explicit vectorization thus requires the programmer to ascertain that the maximum vector length is not exceeded.

Descriptors exist in CYBER 200 FORTRAN as a special data type, and can be assembled and used directly by the programmer. The executable ASSIGN statement corresponds to the PACK instruction, and the descriptor to be assembled must appear in a non-executable DESCRIPTOR statement:

```
DESCRIPTOR AD,BD,CD
. . .
ASSIGN AD,A(1;N)
ASSIGN BD,B(1;N)
ASSIGN CD,C(1;N)
AD = BD + CD
```

This FORTRAN code directly corresponds to the META code discussed above. Each descriptor represents a vector, and each statement (excluding ASSIGN) containing one or more descriptors is, therefore, automatically a vector statement.

Descriptors must either implicitly or by explicit type declaration be declared to have the same data type as the arrays with which they are linked in ASSIGN statements. A descriptor can be defined once and for all in an ASSIGN statement and then be used several times, or it can be defined many times with different values. A given vector statement may contain both explicit and implicit (descriptor) notation in a mixture. For all practical purposes the two notations result in equally

efficient machine code - the overhead saved by defining and reusing descriptors rather than using explicit vector notation is probably buried under vector instructions anyway.

Descriptors are convenient to use, look nice, and save programming effort. To use them is also an excellent way of confusing the reader of your code, since the defining characteristics of the vectors appear only in the ASSIGN statements, and not in the vector statements themselves. Should you nevertheless choose to use the implicit notation, the legibility of your code can be enhanced if you abide by the following rules:

1) Let all descriptor names end with the letter D, and avoid a trailing D for all other variables. If possible, let the descriptor name be the name of the array it's pointing into, and then append a D.

2) Place all ASSIGN statements just prior to the statement(s) in which the descriptors are used. Don't hesitate to redefine a descriptor to the same value several times - clarity is worth a lot.

17.4 V-FUNCTIONS

Almost all of the standard scalar SYSLIB functions, offered in any normal FORTRAN environment, are also available as vector functions on the CYBER 203/205. Those not available in vector mode are COTAN, SINH, COSH, TANH, RANF, SECOND, TIME, DATE plus all functions whose input arguments or value is of data type DOUBLE PRECISION, save for DBLE. With the exception of vector MAX/MIN (section 17.6), the name of the vector version of a given scalar function is simply the scalar name prefixed by a V, and we will refer to them as V-functions. A complete list of them is:

VFLOAT	VAINT	VINT	VIFIX	VABS	VIABS	VSQRT
VSIGN	VISIGN	VMOD	VAMOD	VIDIM	VDIM	
VSIN	VCOS	VTAN	VASIN	VACOS	VATAN	VATAN2
VEXP	VALOG	VALOG10				
VSNGL	VDBLE	VREAL	VAIMAG	VCMPLX	VCONJG	
VCABS	VCSQRT	VCSIN	VCCOS	VCEXP	VCLOG	

The input arguments to a V-function is either one vector (V1), or two vectors (V1 and V2), or one vector and one scalar. Vector

expressions may not be used. The result is always a vector (VR) and we must, therefore, think of the V-functions as vector entities, just as we think of scalar functions as scalar entities. Since a vector represents storage in memory, we must in the syntax of the V-functions also include a pointer to where the result, or its "value-vector", shall reside. That is done by making VR part of the V-function expression itself:

```
VFUNC(V1;VR)
VFUNC(V1,V2;VR)
```

This syntax allows us to treat a V-function expression as a true vector with a data type determined by the data type of VR. Let A and B be real arrays and X complex. Then the following is true:

```
VSIN(A(1;N) ; B(1;N))              is a real vector
VCMPLX(A(1;N),B(1;N) ; X(1;N))     is a complex vector
```

The V-functions can be used as vectors in vector expressions but may not appear as arguments of other V-functions. The most common usage is probably in vector assignment statements where the target vector (left-hand side) coincides with the V-function's value-vector location, as in:

```
B(1;N) = VEXP(A(1;N) ; B(1;N))
```

Had the target vector been different from B(1;N), the value-vector, then a copy to the target vector would have been required after the exponentiation. The "redundancy" in the expression above tells the compiler that such a copy operation is not needed.

We can now explicitly "vectorize" almost all SYSLIB function references - not only those accepted by the automatic vectorizer. The following examples illustrate this.

Scalar(10)

```
      DO 10 J = 1,N
      A(J) = ABS(B(J))
   10 CONTINUE
```

Vector(10)

```
      A(1;N) = VABS(B(1;N) ; A(1;N))
```

Scalar(20)

```
      DO 20 J = 1,N
      B(J) = SQRT(1.0 - COS(A(J))**2)
   20 CONTINUE
```

Vector(20)

```
      B(1;N) = 1.0 - VCOS(A(1;N) ; B(1;N))**2
      B(1;N) = VSQRT(B(1;N) ; B(1;N))
```

17.0 EXPLICIT VECTORIZATION
17.4 V-FUNCTIONS

Scalar(30)

```
      DO 30 J = 1,N
      A(J) = TAN(X(J)) + TAN(Y(J))
   30 CONTINUE
```

Vector(30)

```
      A(1;N)=VTAN(X(1;N);T(1;N))+VTAN(Y(1;N);S(1;N))
```

In Vector(30) the two temporary arrays T and S were necessary to introduce, for obvious reasons.

The vector length at which the V-functions outperform their scalar counterpart is in general quite small - maybe 5 or less. The stream rate is about 3-6 times higher on the CYBER 203, and for most of them significantly more than that on the CYBER 205. Their introduction into your code will therefore almost always give you a substantial speedup. Performing all divides in vector mode and using V-functions wherever possible should be your first goal in terms of explicit vectorization. It may be all you need!

Note that a loop containing a divide or a function reference may often be partially vectorized:

```
      DO 40 J = 1,N
      A(J) = SIN(X(J))/Y(J)
      IF(A(J)) 37,38,39
      . . .
```

This loop is certainly not automatically vectorizable, and it may even be hard to do much the explicit way. However, assuming that arrays A, X, and Y are not unduly tampered with in the latter part of the loop, a partial vectorization is possible by extracting the first line of code. This time we will use descriptors, to illustrate that they can be used interchangeably with explicit vector notation:

```
      DESCRIPTOR AD,XD,YD
      . . .
      ASSIGN AD,A(1;N)
      ASSIGN XD,X(1;N)
      ASSIGN YD,Y(1;N)
      AD = VSIN(XD;AD)/YD
      DO 40 J = 1,N
      IF (A(J)) 37,38,39
      . . .
```

17.5 CONTROL VECTORS

A logical constant or variable occupies one word of storage. This is true on the CYBER 203/205 as well as on most other machines. However, only the rightmost bit is used: 1 for .TRUE. and 0 for .FALSE.. Needless to say, this is a little wasteful - why not pack 64 logical values in each word? Indeed, that is what can be accomplished on the CYBER 203/205 by using the data type BIT. Arrays of type BIT are typically used to control vector operations, and we therefore often talk about control vectors rather than BIT vectors. A descriptor that is to be used to represent a BIT vector must also be declared as data type BIT, as in the following example:

```
DESCRIPTOR BVD
BIT BV(100),BVD
REAL A(100),B(100)
.  .  .
ASSIGN BVD,BV(1;100)
BVD=A(1;100).GT.0.
B(1;100)=Q8VCTRL(A(1;100),BVD;B(1;100))
```

The first vector statement compares each element of A with zero. If A(K).GT.0. the BIT element BV(K) is set to 1, and otherwise it is cleared to 0. That provides us with a string of 100 bits whose value reflects the outcome of the 100 comparisons. Subsequently, in the last line, a Q8V-function is invoked, using A(1;100) as source vector, BVD as control vector, and B(1;100) as target. We will talk more about Q8-functions (Q8V and Q8S) in a later section, but it doesn't hurt to have a preview. Q8VCTRL copies the Kth element of the source vector into the Kth location of the target vector if the Kth bit of the control vector is set, and does nothing otherwise. The vector instructions above thus perform the same work as the following scalar loop, which without the use of a control vector clearly would not have been vectorizable:

```
      DO 10 J = 1,100
      IF (A(J).GT.0.)  B(J) = A(J)
   10 CONTINUE
```

Q8VCTRL handles only real or integer vectors. It can, just like the V-functions, be perceived as a vector of the same data type as its value vector, and may thus appear in vector expressions.

17.0 EXPLICIT VECTORIZATION
17.5 CONTROL VECTORS

Sometimes it can be convenient to compute both desired and undesired results, storing them in a temporary array, and then transfer only the former ones into the proper target array:

Scalar

```
      DIMENSION A(100),X(100)
      DO 10 J = 1,N
      IF (X(J).GT.0.)  A(J) = COS(X(J))
      IF (X(J).LE.0.)  A(J) = SIN(X(J))
   10 CONTINUE
```

Vector

```
      DIMENSION A(100),X(100),T(100)
      BIT BIT(100)
      BIT(1;N) = X(1;N).LE.0.
      A(1;N) = VCOS(X(1;N) ; A(1;N))
      T(1;N) = VSIN(X(1;N) ; T(1;N))
      A(1;N) = Q8VCTRL(T(1;N),BIT(1;N) ; A(1;N))
```

An even more efficient solution would, of course, be to add pi/2 to the elements of X that are positive, and then make only N trigonometric function references rather than 2N:

```
      PIHALF = ACOS(0.)
      BIT(1;N) = X(1;N).LE.0.
      T(1;N) = X(1;N)+PIHALF
      T(1;N) = Q8VCTRL(X(1;N),BIT(1;N) ; T(1;N))
      A(1;N) = VSIN(T(1;N) ; A(1;N)
```

Two Q8V-functions are available to produce cyclic patterns of bits. Q8VMKO creates a bit vector with leading 1's and pads with 0's to the full cycle length, whereafter it repeats the pattern until the length of the value vector (of type BIT) is exhausted. The first argument specifies the number of leading 1's, and the second the cycle length, which need not be a divisor of N. Q8VMKZ works the same way, but starts with leading 0's and pads with 1's:

```
BV(1;N)=Q8VMKO(2,3;BV(1;N))   produces 11011011...
BV(1;N)=Q8VMKZ(2,3;BV(1;N))   produces 00100100...
```

Note that bit vectors can be perceived as strings of logical variables, indicating that they can be used in expressions with logical operators:

```
REAL X(100),Y(100),A(100),B(100)
BIT B1(100),B2(100),B3(100)
. . .
B1(1;N) = X(1;N).GT.Y(1;N)
B2(1;N) = A(1;N).EQ.B(1;N)
B3(1;N) = B1(1;N).OR.B2(1;N)
```

Since an equal sign connects items of the same data type, we must perceive A(1;N).EQ.B(1;N) as a bit vector. The following expression is thus legal:

```
B3(1;N) = B1(1;N).OR.A(1;N).EQ.B(1;N)
```

How do you clear (zero) a bit vector? Well, try this one:

```
B3(1;N) = B3(1;N).XOR.B3(1;N)
```

To conclude this section we will illustrate the use of bit vectors with cyclic patterns by solving the following vectorization problem:

Problem

Let X be dimensioned X(N,N), where N is less than 256, and perform the following operation in vector mode:

```
      DO 10 J = 2,N
      DO 10 K = 1,N
10 X(J-1,K) = X(J-1,K) + X(J,K)
```

A translation to english is: For J=2,N, replace row(J-1) with row(j-1) + row(J).

Solution

First off, we observe that the loops can be switched. Furthermore, there is no recursion, not even with respect to J: X(J-1,K) and X(J,K) are both "old" values, since J is increasing (forward reference).

```
      DO 10 K = 1,N
      DO 10 J = 2,N
10 X(J-1,K) = X(J-1,K)+X(J,K)
```

Now the vector structure is apparent, and the innermost loop is easily vectorized:

```
      DO 10 K = 1,N
   10 X(1,K;N-1) = X(1,K;N-1) + X(2,K;N-1)
```

To clarify what's really happening, let's assume that N=3, and label the elements of X consecutively as 1,2,3,...,9. The work performed by the loop above can now be depicted as follows:

```
     1   2   3   4   5   6   7   8   9
+    2   3   x   5   6   x   8   9   x
-------------------------------------
   1+2 2+3  x  4+5 5+6  x  7+8 8+9  x
```

The picture makes it obvious that if we had a way of excluding every 3rd (Nth) element from the summation, we would only need one vector add. Such a solution is indeed possible:

```
DESCRIPTOR BITD,XD,TD
BIT BIT(N,N),BITD
DIMENSION X(N,N),T(N,N)
.  .  .
L = N*N-1
ASSIGN XD,X(1,1;L)
ASSIGN TD,T(1,1;L)
ASSIGN BITD,BIT(1,1;L)
BITD = Q8VMKO(N-1,N ; BITD)
TD = XD + X(2,1;L)
XD = Q8VCTRL(TD,BITD ; XD)
```

This solution requires only 3 startup times but manipulates 3*(N*N-1) vector elements, while the former requires N startup times and manipulates N*N-N vector elements. For small N, the startup times dominate, while for large N the work performed (proportional to N*N) becomes the timekiller. The bit vector solution is favorable for N less than 142 on the CYBER 203, and for N less than 84 on the CYBER 205. As we shall see later, a third solution is possible (using the special CALL-syntax or the WHERE statement) which will always be faster than performing the additions one column at a time.

17.6 THE WHERE STATEMENT

As a standard feature on both the CYBER 203 and CYBER 205, almost all vector instructions accept an optional control vector. That provides a means of, figuratively, turning a given vector operation on and off according to a specific bit pattern. In reality the operation in question is always performed for each vector element, but a 1 in the Kth position of the control vector says "store the result" while a 0 says "throw it away". We have already seen an example of that in Q8VCTRL, which moves an element from the source vector to the target vector only when the corresponding bit in the control vector is set, and does nothing otherwise.

Some Q8-functions (section 17.7) accept an optional control vector argument, but the syntax for basic arithmetic operations does not permit a similar usage. If the current FORTRAN compiler has release number 1.5.1 or less, then the use of control vectors will in many cases be possible only via the special call-syntax (CALL Q8...) introduced in section 17.8. In Release 1.5.2, however, which is scheduled for fall 1981, the use of control vectors has been greatly facilitated by the introduction of the WHERE statement. The general format of the basic structure is:

```
WHERE (bitexp) vexp
```

The bit expression, bitexp, can be either a bit vector (control vector) in explicit or descriptor form, or any expression that evaluates to a bit vector, such as A(1;N).NE.0., where A is a real or integer array. The vector expression, vexp, can be any vector assignment statement containing one or several of the operators +,-,*,/, and/or one or several of the function references VFLOAT, VIFIX, VINT, VAINT, VSQRT, VABS, VIABS. The only data types allowed are real and integer. Some examples will illustrate the use:

Scalar(10)

```
      DO 10 J = 1,N
      IF (Y(J).NE.0.)  A(J) = X(J)/Y(J)
10 CONTINUE
```

Vector(10)

```
      WHERE (Y(1;N).NE.0.)  A(1;N)=X(1;N)/Y(1;N)
```

Scalar(20)

```
      DO 20 J = 1,N
      IF (Y(J).GE.0.)  X(J) = SQRT(Y(J))
20 CONTINUE
```

Vector(20)

```
BIT BIT(N),BITD
DESCRIPTOR BITD
ASSIGN BITD,BIT(1;N)
BITD = Y(1;N).GE.0.
WHERE (BITD) X(1;N) = VSQRT(Y(1;N);X(1;N))
```

When operations are guided by control vectors in this manner, dividing by zero or taking the square root of a negative number will not cause abortions, since the produced indefinites are thrown away, and therefore ignored.

We can now solve the problem in the end of the previous section more efficiently. In fact, the vector code below always executes faster than when the additions are done a column at a time:

Scalar

```
      DIMENSION X(N,N)
      DO 10 J = 2,N
      DO 10 K = 1,N
10    X(J-1,K) = X(J-1,K) + X(J,K)
```

Vector

```
      BIT BIT(N,N),BITD
      DESCRIPTOR BITD,XD
      DIMENSION X(N,N)
      L = N*N-1
      ASSIGN BITD,BIT(1,1;L)
      BITD = Q8VMKO(N-1,N ; BITD)
      ASSIGN XD,X(1,1;L)
      WHERE (BITD) XD = XD + X(2,1;L)
```

A more complicated situation occurs when the outcome of an IF-test determines which of two blocks of code that should be executed. In such cases the block WHERE and block OTHERWISE syntax is what you need. Two types of structures are permitted:

```
WHERE (bitexp)
  vector statements
END WHERE
```

```
WHERE (bitexp)
  vector statements
OTHERWISE
  vector statements
END WHERE
```

Any number of vector statements may appear in the blocks between the boundary statements WHERE, OTHERWISE, and END WHERE, but they must all abide by the rules for the single vector expression in the WHERE statement defined earlier. All operations in the WHERE block (between WHERE and END WHERE or between WHERE and OTHERWISE) will be performed for all elements, but a result is stored only when the corresponding bit in "bitexp" is set (=1).

Similarly, the statements in the OTHERWISE block (between OTHERWISE and END WHERE) will cause a result to be stored only when the corresponding bit is cleared (=0). When the OTHERWISE statement is present, the WHERE block may be empty. The following is a typical example [the vector version uses a temporary array T(N)]:

Scalar

```
      DO 30 J = 1,N
      Q = A(J) + SQRT(A(J)**2 + B(J))
      IF (A(J).GT.0.)  GOTO 20
      IF (Q.GT.0.)  GOTO 20
      P(J) = A(J)*X(J)*(X(J)-8.)/16
      GOTO 30
20    P(J) = Q/2
30    CONTINUE
```

Vector

```
      T(1;N) = A(1;N)*A(1;N) + B(1;N)
      T(1;N) = VSQRT(T(1;N) ; T(1;N))
      T(1;N) = A(1;N) + T(1;N)
      WHERE (A(1;N).GT.0.  .OR.  T(1;N).GT.0.)
         P(1;N) = .5*T(1;N)
      OTHERWISE
         P(1;N) = (1./16.)*A(1;N)*X(1;N)
         P(1;N) = P(1;N)*(X(1;N)-8.)
      END WHERE
```

17.7 Q8-FUNCTIONS

Many machine instructions performing vector operations that go a little beyond normal arithmetic have been made conveniently accessible to the FORTRAN programmer. That includes all so called Vector Macros, some of which have been mentioned earlier, but also many others. The established convention is that a name prefixed by Q8 indicates that the named function really corresponds to a direct machine instruction, and does not generate a subprogram call. Q8S is used whenever a scalar result is computed, while Q8V means that the result is a vector. The arguments are usually one or two vectors (one of which may be a scalar), and in many cases a control vector may (or must) be specified to govern the operation in question. Q8V-functions follow the same syntax as V-functions: the value vector must be specified as the last "argument", preceded by a semicolon. That is not true for Q8S-functions - specifying storage for a scalar value would be somewhat meaningless. All Q8-functions can be used in expressions; their data types are in general determined by their arguments.

17.0 EXPLICIT VECTORIZATION
17.7 Q8-FUNCTIONS

The table below lists all Q8S-functions. The descriptions are brief, and the manual (Chapter 14) should be consulted for details about usage. AD and BD symbolize vectors of length N, and point into arrays A and B respectively. The data type of A must agree with that of B, and may be either real or integer. X is a scalar of the same data type as A. K, N, and L are integer variables. An optional control vector can be used as an additional, last, argument in Q8SMAX, Q8SMAXI, Q8SMIN, Q8SMINI, Q8SSUM, and Q8SPROD - consult your manual for details.

X = Q8SMAX(AD)	X = MAXIMUM(A(J), J=1,N)
K = Q8SMAXI(AD)	A(K+1) = MAXIMUM(A(J), J=1,N) defines K.
X = Q8SMIN(AD)	X = MINIMUM(A(J), J=1,N)
K = Q8SMINI(AD)	A(K+1) = MINIMUM(A(J), J=1,N) defines K.
X = Q8SDOT(AD,BD)	X = SUM(A(J)*B(J), J=1,N)
X = Q8SSUM(AD)	X = SUM(A(J), J=1,N)
X = Q8SPROD(AD)	X = PRODUCT(A(J), J=1,N)
N = Q8SCNT(BITD)	N = # of 1's in bit vector BITD.
L = Q8SLEN(AD)	L = Length field of AD, converted to integer.
K = Q8SEQ(AD,BD)	(A(K+1),B(K+1)) first pair related by .EQ..
K = Q8SNE(AD,BD)	(A(K+1),B(K+1)) first pair related by .NE..
K = Q8SGE(AD,BD)	(A(K+1),B(K+1)) first pair related by .GE..
K = Q8SLT(AD,BD)	(A(K+1),B(K+1)) first pair related by .LT..
Q8SEXTB	Extracts bits from a word - see manual.
Q8SINSB	Inserts bits into a word - see manual.
Q8SDFB	Tests data flag branch register - see manual.

There are 10 Q8V-functions that perform some type of data motion, and 11 others doing miscellaneous work. We will first give the "calling sequences" to all of them. AD and BD symbolize real or integer source vectors, one of which may be a scalar. The arrays pointed into, A and B, must agree with each other in data type. X and Y are both either real or integer scalars while L, K1, K2, and N are integers. IXD is an integer index vector, ZD is a control vector, and CD is a value vector (result) which agrees in data type with the first input argument - except in Q8VMKO and Q8VMKZ, where CD is of type BIT.

17.0 EXPLICIT VECTORIZATION
17.7 Q8-FUNCTIONS

Data Motion	Others
CD = Q8VCMPRS(AD,ZD;CD)	CD = Q8VINTL(X,Y;CD)
CD = Q8VCTRL (AD,ZD;CD)	CD = Q8VMKO (K1,K2;CD)
CD = Q8VGATHP(AD,L,N;CD)	CD = Q8VMKZ (K1,K2;CD)
CD = Q8VGATHR(AD,IXD;CD)	CD = Q8VADJM(AD;CD)
CD = Q8VMASK (AD,BD,ZD;CD)	CD = Q8VAVG (AD,BD;CD)
CD = Q8VMERG (AD,BD,ZD;CD)	CD = Q8VAVGD(AD,BD;CD)
CD = Q8VREV (AD;CD)	CD = Q8VDELT(AD;CD)
CD = Q8VSCATP(AD,L,N;CD)	IXD = Q8VEQI (AD,BD;IXD)
CD = Q8VSCATR(AD,IXD;CD)	IXD = Q8VGEI (AD,BD;IXD)
CD = Q8VXPND (AD,ZD;CD)	IXD = Q8VLTI (AD,BD;IXD)
	IXD = Q8VNEI (AD,BD;IXD)

A detailed description of how all of these Q8V-functions work appears in chapter 14 of your FORTRAN manual, and we will here only give a brief summary. Whenever a control vector (ZD) appears in the argument list, we will use .T. and .F. to indicate what happens when the Kth bit, Z(K), is set and cleared respectively. In these cases, I and J are assumed to have the values 1 initially.

```
Q8VCMPRS    .T.: C(J)=A(K), J=J+1 / .F.: no action.
Q8VCTRL     .T.: C(K)=A(K) / .F.: no action.
Q8VGATHP    (C(J)=A(1+(J-1)*L), J=1,N)
Q8VGATHR    (C(J)=A(IX(J)), J=1,N)
Q8VMASK     .T.: C(K)=A(K) / .F.: C(K)=B(K)
Q8VMERG     .T.: C(K)=A(I), I=I+1 / .F.: C(K)=B(J), J=J+1
Q8VREV      (C(J)=A(N+1-J), J=1,N)
Q8VSCATP    (C(1+(J-1)*L)=A(J), J=1,N)
Q8VSCATR    (C(IX(J))=A(J), J=1,N)
Q8VXPND     .T.: C(K)=A(I), I=I+1 / .F.: C(K)=0 or 0.0

Q8VINTL     (C(J)=X+(J-1)*Y, J=1,N)
Q8VMKO      Cyclic bit pattern: K1 1's and (K2-K1) 0's
Q8VMKZ      Cyclic bit pattern: K1 0's and (K2-K1) 1's
Q8VADJM     (C(J)=(A(J)+A(J+1))/2, J=1,N-1)
Q8VAVG      (C(J)=(A(J)+B(J))/2, J=1,N)
Q8VAVGD     (C(J)=(A(J)-B(J))/2, J=1,N)
Q8VDELT     (C(J)=(A(J)-A(J+1))/2, J=1,N-1)
Q8VEQI      (IX(J) = Q8SEQ(A(J),BD), J=1,N)
Q8VGEI      (IX(J) = Q8SGE(A(J),BD), J=1,N)
Q8VLTI      (IX(J) = Q8SLT(A(J),BD), J=1,N)
Q8VNEI      (IX(J) = Q8SNE(A(J),BD), J=1,N)
```

17.0 EXPLICIT VECTORIZATION
17.7 Q8-FUNCTIONS

As you can see there is a vast body of Q8-functions available, many of which probably appear to perform somewhat peculiar tasks. Although at this point their usefulness may seem questionable, you will, no doubt, find that your appreciation increases gradually as you become more proficient in explicit vector programming. On the CYBER 205 all of them represent significant time savings over scalar code, while that is true only for some on the CYBER 203.

The execution time for a Q8-function can be expressed as S+R*N, where S is a startup time, and R is a constant of proportionality. For Q8S-functions N is the length of the input vector(s), while for Q8V-functions it is the length of the result vector, CD. For non-iterative search functions, such as Q8SMAX and Q8SEQ, N represents the number of elements actually searched. For iterative search functions, such as Q8VEQI, the execution time must be expressed as S+(S'+R'*NN)*N, where NN is the average number of elements searched per iteration, and N is the length of AD and IXD. In these cases, thus, R=S'+R'*NN.

The startup times can vary quite a bit from case to case, and it would be misleading to quote any "exact" values. As a general rule, though, they lie in the range 150-350 cycles on the CYBER 203, and 50-150 cycles on the CYBER 205. The constant of proportionality, R, does in general not fluctuate very much, and we will therefore qoute typical and approximate R-values in the table below. Two R-values are given for each function: the first refers to the CYBER 203, and the second to the CYBER 205. As an example, the values given for Q8VGATHR are 40 and 5/4, which means that the time required to gather a vector of length N is 40N cycles on the CYBER 203, and 1.25N cycles on the CYBER 205 - plus relevant startup times. Remember: the timings are typical and approximate!

Name	203	205	Name	203	205	Name	203	205
Q8XMAX	13	1	Q8VCMPRS	2	1/2	Q8VINTL	2	1
Q8SMAXI	13	1	Q8VCTRL	1	1/2	Q8VMKO	1/8-2	1/16-1
Q8SMIN	13	1	Q8VGATHP	-	5/4	Q8VMKZ	1/8-2	1/16-1
Q8SMINI	13	1	Q8VGATHR	40	5/4	Q8VADJM	1	1/2
Q8SDOT	12	1	Q8VMASK	10	1/2	Q8VAVG	1	1/2
Q8SSUM	11	1	Q8VMERG	8	1/2	Q8VAVGD	1	1/2
Q8SPROD	12	1	Q8VREV	8	1/2	Q8VDELT	1	1/2
Q8SCNT	1/8	1/16	Q8VSCATP	-	5/4	Q8VEQI	160+NN	62+NN/2
Q8SEQ	1	1/2	Q8VSCATR	38	5/4			
			Q8VXPND	8	1/2			

17.8 SPECIAL CALL SYNTAX

At this point you probably feel quite saturated with all the special CYBER 200 FORTRAN features. Is there really no end to it? Well, for the average FORTRAN programmer there is, and if you belong to that category you may well skip this section. We (=the author of this document) can only think of two situations that would justify your hanging in there a little longer: either you enjoy trying to fool the compiler with some extra tricky turns, or Release 1.5.2 of the FORTRAN compiler is not yet made current, and you need the power of the WHERE statement. In the former case you are probably in for some grief, unless you have substantial experience as an assembly language programmer. In the latter case, the situation is not that serious, since the rest of this section will focus on how to simulate the WHERE statement - if you pay attention and exactly follow the "perscription" you should be alright.

The Special Call Syntax has been invented to enable you to insert any machine instruction directly into the compiler generated machine code. FORTRAN triggers on the 7 characters "CALL Q8", whose only use should be to prefix mnemonics borrowed directly from META, the CYBER 200 assembly language. As an example, the floating point addition "A=B+C" would in META be coded as "ADDN B,C,A", but could in FORTRAN be expressed as "CALL Q8ADDN(B,C,A)".

A very meager writeup of the usage of the Special Call Syntax can be found on pages 13.1-2 in your FORTRAN manual. It is complemented by a complete table of the "calling sequences" in Appendix D, where you can also find some additional information. The hardware manual, however, is the only place where you can find an adequate description of how each machine instruction really works - and that is something you need to know if you plan to create your own tricky code. The WHERE statement simulators, though, should find sufficient information below.

The WHERE statement allows you to use a bit vector to control the operations +,-,*,/, and also in conjunction with the function references VFLOAT, VAINT, VIFIX, VINT, VABS, VIABS, and VSQRT. To accomplish the same you must first break down the vector expression that you wish to have evaluated under the control of a bit vector into a series of (dyadic) vector statements, each containing only one arithmetic operation or function reference. Each such statement can then be expressed as one or a few CALL Q8

statements, according to the table below. Let's first define the conventions and set some rules.

All machine instructions of the type considered have the same format, and will in FORTRAN appear as

```
CALL Q8xxxxx(G,X,A,Y,B,Z,C)
```

where xxxxx symbolizes the META mnemonic. All seven fields must be defined by the comma separators, but some of the arguments may be omitted. All arguments, except G, represent registers, and must always be specified with variables - never constants. The G-designator, if not omitted, should appear as a hexadecimal constant in the format X'nn', where nn is a two-digit hexadecimal number. The meaning and use of each argument (designator) is for our purposes as follows:

G This is an instruction modifier, and should be omitted except when one of the two source operands is a scalar. X'10' should be used when A is a scalar (broadcast A), and X'08' when the same is true for B.

X Should always be omitted.

A This is the first source operand. If A is a vector, it must be real or integer, and can be specified either with the explicit notation, e.g. A(1,1;N), or by using a descriptor, e.g. AD or KAD. No vector expressions are allowed. A scalar to be broadcast is specified with a real or integer scalar variable, and requires X'10' in the G-field.

Y Should always be omitted.

B This is the second input operand, which sometimes will be omitted. The rules are the same as those for A, except that a scalar value requires the G-field to be X'08'.

Z This is the control vector. An explicit vector or a descriptor of data type BIT should appear in this field. Note that an expression of type A(1;N).LE.B(1;N) is not permitted - the resulting bit vector must be created in a separate vector statement.

C This is the result vector, and should be specified analogous to A and B, except that a scalar value here makes no sense.

We will use the descriptor notation and symbolize the source vector with AD and BD when they are real, while KAD and KBD implies that they are of data type integer. Similarly, ZD (of type BIT) will be used for the control vector, and CD or KCD for the result vector. S is a real scalar while KS is an integer. In the case of the multiplication of two integer vectors, for which no direct machine instruction exists, a scratch vector QQD, of arbitrary data type, has been used.

FORTRAN without Control vector	FORTRAN with Control vector (ZD)
CD = AD+BD	CALL Q8ADDNV(, , AD, , BD,ZD, CD)
CD = AD+S	CALL Q8ADDNV(X'08', , AD, , S ,ZD, CD)
CD = S+BD	CALL Q8ADDNV(X'10', , S , , BD,ZD, CD)
CD = AD-BD	CALL Q8SUBNV(, , AD, , BD,ZD, CD)
CD = AD*BD	CALL Q8MPYSV(, , AD, , BD,ZD, CD)
CD = AD/BD	CALL Q8DIVSV(, , AD, , BD,ZD, CD)
KCD = KAD+KBD	CALL Q8ADDXV(, ,KAD, ,KBD,ZD,KCD)
KCD = KAD-KBD	CALL Q8SUBXV(, ,KAD, ,KBD,ZD,KCD)
KCD = KAD*KBD	CALL Q8ADDNV(X'08', ,KAD, , ,ZD,QQD) CALL Q8ADDNV(X'08', ,KBD, , ,ZD,KCD) CALL Q8MPYSV(, ,QQD, ,KCD,ZD,KCD) CALL Q8TRUV (, ,KCD, , ,ZD,KCD)
KCD = S*KBD	CALL Q8ADDNV(X'08', ,KBD, , ,ZD,KCD) CALL Q8MPYSV(X'10', , S , ,KCD,ZD,KCD) CALL Q8TRUV (, ,KCD, , ,ZD,KCD)
KCD = KAD/KBD	CALL Q8DIVUV(, ,KAD, ,KBD,ZD,KCD) CALL Q8TRUV (, ,KCD, , ,ZD,KCD)
KCD = S/KBD	CALL Q8ADDNV(X'08', ,KBD, , ,ZD,KCD) CALL Q8DIVSV(X'10', , S , ,KCD,ZD,KCD) CALL Q8TRUV (, ,KCD, , ,ZD,KCD)

17.0 EXPLICIT VECTORIZATION
17.8 SPECIAL CALL SYNTAX

```
CD  = VFLOAT(KAD;CD)     CALL Q8ADDNV(X'08', ,KAD, ,   ,ZD, CD)

CD  = VAINT(AD;CD)       CALL Q8TRUV (     , , AD, ,   ,ZD, CD)
                         CALL Q8ADDNV(X'08', , CD, ,   ,ZD, CD)

KCD = VIFIX(AD;KCD)      CALL Q8TRUV (     , , AD, ,   ,ZD,KCD)
                         KS=0
                         CALL Q8ADJEV(X'08', ,KCD, , KS,ZD,KCD)

CD  = VABS(AD;CD)        CALL Q8ABSV (     , , AD, ,   ,ZD, CD)

KCD = VIABS(KAD;KCD)     CALL Q8ABSV (     , ,KAD, ,   ,ZD,KCD)

CD  = VSQRT(AD;CD)       CALL Q8SQRTV(     , , AD, ,   ,ZD, CD)
```

In a WHERE statement or a WHERE block, results are stored "on 1's" and discarded "on 0's". The same is the case in all Special Calls above. In an OTHERWISE block, however, the situation is reversed: results are stored "on 0's" and discarded "on 1's'. To accomplish the same using the Special Call Syntax, the G-designator must be changed. The recipe is to add hex 40 to the value already used: a previously blank G-field becomes X'40', X'08' changes to X'48' and X'10' to X'50'.

It should be emphasized that the Special Call Syntax really isn't a part of FORTRAN. It's more appropriate to describe it as a form of the assembly language META which the FORTRAN compiler has learned to understand. As a consequence, you are essentially in no man's land whenever you use it. Very little, if any, checking of the correctness of the CALL Q8 statements will take place - the compiler couldn't care less, for instance, if you use wrong data types or forget to properly declare the descriptors.

APPENDIX A - REFERENCE MANUALS

The manuals listed below can be ordered through your CDC sales representative, or directly from:

Literature Distribution Services
STP005
308 North Dale Street
St. Paul, Minnesota 55102

The SCOPE Front-End

CYBERNET Services CYBER 200 CYBERNET Center Users Guide	84001330
CYBERNET Services SCOPE 3.4 Reference Manual	84000021
COMSOURCE INTERCOM 5 Reference Manual	60456510
CYBERNET Services UPDATE Reference Manual	84000016

The NOS Front-End

CYBERNET Services CYBER 200 NOS Front-End Users Guide	84001990
CYBERNET Services NOS, Volume 1 (Tutorial)	84000320
CYBERNET Services NOS, Volume 2 (Interactive Usage)	84000360
CYBERNET Services NOS, Volume 3 (Comprehensive Usage)	84000370
CYBERNET Services XEDIT Reference manual	76071000
CYBERNET Services UPDATE Reference Manual	84000016

The CYBER 203/205

CYBER 200 FORTRAN Version 3	60457040
CYBER 200 Operating System Version 1, Volume 1 of 2	60457000

The CYBER 203/205 - Advanced Usage

CYBER 200 Assembler Version 3	60457050
CYBER 200 Operating System Version 1, Volume 2 of 2	60457010
CYBER 203 Hardware Reference Manual	60256010
CYBER 205 Hardware Reference Manual	60256020

APPENDIX B - PHONE NUMBERS AND OPERATING HOURS (C.S.T.)

CYBER 74 - NOS

300 baud: (612) 482-6712
4800 baud: (800) 328-9693 (612) 482-6710

300 baud-lines to the CYBER 74 are available through the Cleveland Data Services Network. Consult the CYBERNET Network Access Guide (Pub. no. 201,603E) for local telephone numbers (your CDC sales representative should have one).

Operating hours are 7 days a week, 02.00-20.00.

CYBER 175 - SCOPE 3.4

300 baud: (800) 328-9602 (612) 482-5734
4800 baud: (800) 328-9606 (612) 482-4832 (612) 482-4075

The CYBER 175 is also accessible via CYBERLINK from the CYBER 176 in Houston (KHE) - see SYSBULL 18 (Appendix C).

Operating hours are MON, TUE, THU 00.00-20.30, 22.30-23.45 and WED, FRI, SAT, SUN 00.00-23.45.

CYBER 203

Accessed through either one of the two front-ends.
Operating hours are MON-FRI 07.00-20.00 and SAT 12.00-16.00

Customer Services

The HOTLINE is open MON-FRI 08.00-17.00.
(800) 328-9605 (612) 482-2830

APPENDIX C - CYBER 74/CYBER 203 INFORMATION

Important information pertinent to the CYBER 203 user who interfaces via the NOS front-end is available through use of the NOS EXPLAIN command - either interactively or in your NOS batch-job. To obtain a list of the available topics (which appears below), use the following command:

```
EXPLAIN,CYBER 200.
```

To obtain the information in any one of the topics below, use the following format:

```
EXPLAIN,CY200 topicname.
(example: "EXPLAIN,CY200 ASSIST.")
```

TOPIC	CONTENTS
ASSIST	Service center phone numbers.
HOURS	Service hours.
ACCESS	NOS front-end access phone numbers.
COMMANDS	CYBER 203 interface commands.
APPLS	Available NOS front-end applications.
MANUALS	Relevant CYBER 203 manuals.
FTNUTIL	See Appendix F.
ARCHIVE	Rules for permanent file archiving.
PROBLEMS	Current known problems.
INTERACT	Interactive LOGON info for the CYBER 203.
NEWS	Whatever is new.

APPENDIX D - CYBER 175/CYBER 203 INFORMATION

System bulletins (SYSBULL) are maintained for informing CYBER 203 users of temporary changes and updates to the operating system procedures. The SYSBULL contents is relevant to both the CYBER 175 front-end and the CYBER 203, but not to the CYBER 74 front-end. The bulletins are numbered 1 through 999 and may, or may not, contain information. When the information is outdated, it will be removed. An index is maintained, which reflects the last modification date of each non-empty bulletin.

The SYSBULL control card may be inserted into the current SCOPE job-stream without making a separate run, and should have one of the following formats:

SYSBULL,BATCH.	Gets only the "BATCH" SYSBULL. This automatically (unsolicited) comes out on the first pages of each batch-job, and contains important notices.
SYSBULL,INDEX.	Gets only the index to all SYSBULLs.
SYSBULL,ALL.	Gets all SYSBULLs (including "BATCH" and "INDEX").
SYSBULL,xx.	Gets only SYSBULL number xx (1-3 digit integer).
SYSBULL,xx,yy,zz.	Gets only SYSBULLs numbered xx, yy and zz.

The following batch-job obtains all bulletins:

```
JOB.
USER,username,password.
SYSBULL,ALL.
6/7/8/8
```

NOTE: Users are held responsible for information published in the system bulletins. It is therefore recommended that each time a new bulletin appears, all users request a copy for future reference.

APPENDIX E - CONVERSION AID PROGRAM

The manual that describes the FTN4-5 conversion aid program is "FORTRAN Extended Version 4 to FORTRAN Version 5 Conversion Aid Program" with publication number 60483000. Below follows a description of how to use it, and the intent is that you should be able to manage without acquiring the manual. Note that the program is installed on the NOS front-end (CYBER 74) only.

The program is accessed via the control statement F45, which can be used with a number of parameters. We will here assume the following situation: your code resides on file SRC4, as 80-column card images; you want the converted program, in a form suitable for direct input to the compiler, on file SRC5; you want an output listing consisting of a copy of SRC4 and SRC5 plus related Conversion Aid messages; you want all machine-dependent statements flagged on the output listing. If all of the above is true, then the following format will be adequate:

```
F45,I=SRC4,P=SRC5,PO=F,LO,MD.
```

Other names instead of SRC4 and SRC5 can of course be used. If ",I=SRC4" is omitted, the program assumes "I=INPUT". If ",P=SRC5" is omitted, no converted file will be produced. If ",LO" is omitted, the listing produced will only contain translated and added lines, lines requiring manual action, and Conversion Aid messages and error diagnostics. If ",MD" is omitted, machine-dependent statements will not be flagged.

With an UPDATE generated COMPILE file as input ("I" or "I=COMPILE") you may request an output file (called MODS below) consisting of the UPDATE modifications necessary to accomplish the desired conversion. In that case "PO=F" must be replaced by "PO", and the CI-parameter must be used to specify correction ident (chosen as FIX below):

```
F45,I,P=MODS,CI=FIX,PO,LO,MD.
```

The following order-dependent steps need to be taken to accomplish a successful conversion:

1) Remove all blank lines (if any) preceding continuation lines.

2) Manually convert the following items, as listed in chapter 3 of this document:

   3.1 e-f,h,k,m 3.2 a-b,d-e 3.3 a-b,d,h-k 3.5 a,c-h

   This step may be performed equally well after the next step.

3) Invoke the FTN4-5 Conversion Aid Program, as described above (could be step 2 instead).

4) Manually do the conversions listed below, paying attention to the partial changes already performed by the Conversion Aid:

   3.2h The Conversion Aid have changed the format of octal constants from 10B to O"10". Both formats are illegal, and must be removed.

   3.3f

   3.4a

   3.4b The Conversion Aid has given all ENTRY statements the same argument list as that of the corresponding SUBROUTINE or FUNCTION statement. This must be changed only if the ENTRY points are called with a different number of arguments.

   3.4c The Conversion Aid puts * instead of & for multiple RETURNs in CALL statements. Change appropriately.

   3.4e The Conversion Aid inserts the statement CALL GOTOER after each computed GOTO. Replace with something like STOP'message'.

   3.5b The Conversion Aid inserts END=n in each READ statement, where n is the statement label of the next executable statement. If that statement doesn't have a label, the Conversion Aid gives it one. The action is probably adequate, unless you use the EOF-function.

5) The Conversion Aid changes the Hollerith format nRstring to R"string", and RANF(DUMMY) to RANF(). It also changes "READ fmt,io-list" to "READ (*,fmt,END=n) io-list", and inserts "BZ," in the beginning of all FORMAT statements. All these conversions must be reversed.

6) Change machine-dependent statements to their proper form (read Chapter 4 first). Use of the MD parameter on the F45 card causes the Conversion Aid to flag these items.

APPENDIX F - CYBER 203 POOL FTNUTIL

FTNUTIL is a pool, residing on the CYBER 203, to which all users have access. It contains one controllee file (REQTEMP) and four libraries (PLOTLIB, MSSLIB, QQLIB, LINPACK).

REQTEMP can be used to allocate file space on scratch packs.

PLOTLIB contains routines that simulate the UNIPLOT routines.

MSSLIB contains SPY (a FORTRAN callable routine that measures execution time distribution in your code) and a set of simulated random I/O routines (OPENMS, STINDX, WRITMS, READMS, CLOSMS, RTNFILE).

QQLIB contains a number of FORTRAN callable utility routines (dot products, gather/scatter, etc.) and mathematical algorithms (solvers, eigenvalue finders, inverters, etc.) especially written for the CYBER 203. They are all highly optimized (vectorized when applicable).

LINPACK is a collection of linear algebra routines, as featured on several other computer systems. The package contains both real and complex versions, and the routines perform such functions as decomposition, factorization and solution of general, banded or symmetric matrices. LINPACK also contains a computation kernel (about 20 small routines, essentially performing vector operations) which is often referred to as the BLAS.

To obtain a complete documentation of REQTEMP, PLOTLIB and MSSLIB, plus a more detailed description of the content of QQLIB and LINPACK, please run the following front-end job:

SCOPE

```
JOB.
USER,....
ATTACH,FTNUTIL,ID=FTNUTIL.
COPY,FTNUTIL,OUTPUT.
```

NOS

```
JOB.
USER,....
ATTACH,FTNUTIL/UN=DOCUMAA,NA.
COPY,FTNUTIL,OUTPUT.
```

To obtain a complete documentation of either QQLIB or LINPACK, please run the job above, but first replace FTNUTIL (all occurrences) with either QQLIB or LINPACK. The LINPACK documentation (about 140 pages) can be halved in size by using COPYBR instead of COPY, thereby omitting the documentation of the complex routines.

COMMENT SHEET

MANUAL TITLE: CYBER 203/205 User Guide

PUBLICATION NO.: 84002390 **REVISION:** A

NAME: _____

COMPANY: _____

STREET ADDRESS: _____

CITY: _____ **STATE:** _____ **ZIP CODE:** _____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

☐ **Please reply** ☐ **No reply necessary**

CUT ALONG LINE

AA3419 REV. 4/79 PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION
Headquarters Publications Writing
Publications and Graphics Division
P.O. Box 0, HQC02C
Minneapolis, Minnesota 55440

CUT ALONG LINE

FOLD

FOLD

IMPORTANT REGULATORY NOTICE

Users of Control Data services should be aware that the rules and regulations of the United States and International Telecommunications Regulatory Agencies prohibit Control Data from using communications services it leases from domestic, international and foreign communications carriers to transmit information for its users which is not part of a "single integrated" data processing service. All information transmitted must be directly related to the data processing applications or service provided by Control Data and unprocessed information shall not be allowed through the service between user terminals, either directly or on a store and forward basis. Noncompliance with these rules and regulations may force Control Data to discontinue the users' data processing service.

CORPORATE HEADQUARTERS
8100 34TH AVENUE SOUTH
MINNEAPOLIS, MINNESOTA
MAILING ADDRESS • BOX 0, MPLS., MINN. 55440

SALES OFFICES AND SERVICE CENTERS
IN MAJOR CITIES THROUGHOUT THE WORLD