RECEIVED BY DTIE OCT 13 1967

OAK RIDGE NATIONAL LABORATORY

operated by

UNION CARBIDE CORPORATION
NUCLEAR DIVISION

for the

U.S. ATOMIC ENERGY COMMISSION

ORNL - TM - 1998

COPY NO. - 10

DATE - September 25, 1967

BRANCHING IN FORTRAN

J. G. Sullivan

Abstract

The execution time of Fortran programs can be improved by properly considering both the method used to cause a program branch and the position of the branch point. The discussion points out some general rules which can be applied to Fortran programs which will improve execution time.

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

NOTICE This document contains information of a preliminary nature and was prepared primarily for internal use at the Oak Ridge National Laboratory. It is subject to revision or correction and therefore does not represent a final report.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

LEGAL NOTICE

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Commission, nor any person acting on behalf of the Commission:

- A. Makes any warranty or representation, expressed or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or
- B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Commission" includes any employee or contractor of the Commission, or employee of such contractor, to the extent that such employee or contractor of the Commission, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Commission, or his employment with such contractor.

Branching in Fortran

There are five methods available to the Fortran programmer for choosing among paths for program execution. Four of them are, essentially, performing a test of some variable and, depending on its value, branching to alternate portions of the code. One, the GO TO n type statement where n is a statement number (e.g., GO TO 273), is a branch with no test. The burden of this discussion will be the implications of statements which test and branch and the considerations necessary for the Fortran programmer to efficiently choose among the abundance of possibilities.

The four methods are:

- 1) "Computed" GO TO ($n_1, n_2, \dots, n_j$), I; e.g., GO TO(1,2,3,40,7), I
- 2) "Assigned" GO TO; e.g., GO TO I, where I has been previously defined by an ASSIGN statement.
- 3) Arithmetic IF; e.g., IF(A-B) 1, 2, 3
- 4) Logical IF; e.g., IF(A.GT.B) GO TO 100

As methods 1 and 2 are undoubtedly rarer in occurrence than methods 3 and 4, a few remarks will be made about them before devoting the bulk of the discussion to methods 3 and 4.

GO TO STATEMENTS

First of all, the "assigned" GO TO does not differ logically from the "computed" GO TO statement.

Example 1a:

```
ASSIGN 200 TO ITRA
IF (X-5.) 10, 20, 30
10 CONTINUE
```

- FORTRAN STATEMENTS-

```

      ASSIGN 50 TO ITRA
      GO TO 40
20  CONTINUE
      - FORTRAN STATEMENTS -
      ASSIGN 73 TO ITRA
      GO TO 40
30  IF(X-6.) 31, 40, 40
31  CONTINUE
      - FORTRAN STATEMENTS -
      ASSIGN 371 TO ITRA
40  GO TO ITRA

```

where the execution of statement 40 will cause a transfer of execution to one of the following statements: 20, 50, 73, or 371.

The same coding, but using the "computed" GO TO, is illustrated in the following coding:

Example 1b:

```

      ITRA = 1
      IF(X-5.) 10, 20, 30
10  CONTINUE
      - FORTRAN STATEMENTS -
      ITRA = 2
      GO TO 40
20  CONTINUE
      - FORTRAN STATEMENTS -
      ITRA = 3
      GO TO 40
30  IF(X-6.) 31, 40, 40

```


31 CONTINUE

- FORTRAN STATEMENTS -

ITRA = 4

40 GO TO (200,50,73,371), ITRA

There is no logical difference in this usage but the use of the "computed" GO TO probably makes the coding a little easier to follow. One advantage for choosing the "computed" form over the "assigned" form is that, during the debugging phase, the "assigned" GO TO can give peculiar (and nonreproducible) errors if there is a GO TO ITRA and this statement is executed prior to an ASSIGN TO ITRA. Also, some "programmers" are prone to write arithmetic assignment statements (ITRA = 200) when they should have written ASSIGN 200 TO ITRA. One disadvantage with using the "computed" GO TO during debugging is that while the action taken at statement 40 in Example 1b is clearly defined for ITRA = 1,2,3, or 4 various compilers react in differing ways for ITRA < 1 and ITRA > 4. One's predilections must be the determining factor here. So far as speed of execution is concerned the use of the "assigned" GO TO is definitely faster. This is evident from considering that the "computed" GO TO must either do IF-like tests, e.g.:

IF(ITRA-2) 200, 50, 999

999 IF(ITRA-4) 73, 371, ...

or else treat the addresses of the branch points as if they were an array subscripted by ITRA. There also may be some "mickey-mouse" in taking care of values of ITRA outside the range of transfers. The "assigned" GO TO is never more than a load of ITRA and a branch to the address specified by the value of ITRA. One might also note that if there are only three branch points: GO TO (200,50,73), ITRA the IF(ITRA-2)200,50,73 way of writing it is probably a shade faster in execution. All in all, use of the "computed" GO TO is

possibly better during the debugging phase and the "assigned" GO TO is better in the execution phase. (This, of course, only applies when methods 1 and 2 are used interchangeably as in Example 1.) If one, say, starts off a program with:

READ 1, ITRA

GO TO (1, 2, 3, 4), ITRA

then use of the "assigned" GO TO is, of course, not applicable.

IF STATEMENTS

The logical IF and the arithmetic IF can, in many instances, be rather freely interchanged:

IF (A-B) 1, 2, 2 or IF(A.GE.B) GO TO 2

1 CONTINUE

1 CONTINUE

However, because of the greater complexity of the logical IF each method will be taken up separately.

Method 3.

ARITHMETIC IF

In this method there are, of course, two items for consideration:

1) the expression to be evaluated and tested and 2) the possible branch points.

Whenever any arithmetic operations are called for in the evaluation of an expression (e.g., A-B) the action taken is essentially the same as that performed on the right-hand side of the arithmetic replacement statement except that no storing of the result occurs and the result is tested in whatever way the computer hardware permits. When no arithmetic operation appears, then the manner in which the test of the variable in storage is made is a function of computer hardware. (Some computers can test the variable in memory directly; others may have to load the variable into an accumulator or register of some kind and then perform the test). For illustrative purposes the 360 hardware

operations are used to describe the action taken and the mnemonics to be used are defined as follows:

MNEMONIC, Variable	Action
LOAD,X	The variable, X, is loaded into the register
ADD,X	The variable, X, is added to the register
SUB,X	The variable, X, is subtracted from the register
B,l	Branch to statement l
BZ,l	Branch if the register is zero to statement l otherwise, proceed to the next instruction
BP,l	Branch if the register is greater than 0 to statement l, otherwise proceed.
BM,l	Branch on minus, etc.
BPZ,l	Branch on greater than or equal to zero, etc.
BMZ,l	Branch on minus or zero, etc.
BPM,l	Branch on greater than zero or less than zero, etc.
TEST	Test the register

It is necessary to note that after an addition or subtraction the register can be tested and the branches made without the necessity for a TEST instruction. After a LOAD, a TEST instruction must be given before the branch can be made. (Statement numbers will appear at the left of the mnemonic code).

EXAMPLE: LOAD,X	Explanation: Y is subtracted from X
SUB,Y	and if the result is zero a
BZ,l	branch to statement l is taken.
EXAMPLE: LOAD,X	Explanation: X is loaded, tested, and
TEST	if X is less than zero a branch to
BM,l	statement l is taken.

EXAMPLE: LOAD,X Explanation: WRONG. The TEST must be
 BM,1 given after the LOAD as in the
 preceding example.

Example 2

9 IF (A-B) 1, 2, 3

4 CONTINUE

TIMINGS in μ sec for 360/75 I

9:LOAD,A	.7
SUB,B	.7
BM,1	1.0
BZ,2	1.0
BP,3	1.0

Finding the amount of computer time to execute statement 9 is as follows:

Case	Time (μ sec)
A < B	2.4
A = B	3.4
A > B	4.4

Thus, considerable difference in execution time can occur. In fact, the execution time for the case where A > B is almost double that for the case where A < B. Compilers have no information upon which to base the order in which the tests would be "most optimally" performed and will usually translate this IF in a mechanical way (BM, BZ, EP) or (BP, BZ, BM), etc. Even if the programmer knows that A is usually greater than B, he must also know the order in which the compiler will set up the tests. There is no logical difference in the following rearrangement of statement 9:

9: IF(B-A) 3, 2, 1

yet this arrangement might take about half the time of the statement as originally written!

Take the most extreme example:

Example 3

```
9  IF(A) 1, 2, 3
4  CONTINUE
```

Timing (μ sec)

9: LOAD,A	.7
TEST	.4
BM,1	1.0
BZ,2	1.0
BP,3	1.0

Here the time taken for $A < 0$ is 2.1μ sec and the time taken for $A > 0$ is 4.1μ sec, a considerable difference.

A complication (either helpful or harmful so far as the total execution time) appears when one of the branch points immediately follows the IF. For example:

Example 4

```
9  IF(A-B) 1, 2, 3
2  CONTINUE
```

This could be translated:

Example 4a.

9: LOAD,A	.7
SUB,B	.7
BM,1	1.0
BZ,2	1.0
BP,3	1.0
2: etc.	

or

Example 4b.

9:	LOAD,A	.7
	SUB,B	.7
	BM,1	1.0
	BP,3	1.0

2: etc.

Timings (μ sec)

Example: 4a. 4b.

A < B 2.4 2.4

A = B 3.4 3.4

A > B 4.4 3.4

In most cases, compilers make some use of the fact that the following statement may be one of the IF statement branch points, so a gain in execution time (or, at least, no loss) is possible if a branch point of an IF immediately follows the IF statement.

In order to see how difficult life can be, make the following assumption the A > B branch is most likely. If we write:

Example 5

```

9  IF(B-A) 2, 2, 1
2  CONTINUE

```

which translated, might be:

	LOAD,B	.7
	SUB,A	.7
	BZ,2	1.0
	BP,1	1.0
3	etc.	

The time to reach statement 3 is 3.4 μ sec.

If we were to move statement 3 away from the IF test:

```
Example 5a      9  IF(B-A) 3, 2, 1
                  4  CONTINUE
```

and, if the compiler makes the minus test first, the most frequently followed path is performed in 2.4 μ sec.

In many cases two of the branch points of the arithmetic IF may be identical:

```
IF(A-B) 1, 2, 1
```

This is generally translated as:

```
LOAD,A
SUB,B
BPM,1
BZ,2
```

The same sort of consideration holds as in the discussion above where the branch points were different. One test has still to be performed first and a proper arrangement of statement locations (either following the IF or not following the IF) must be undertaken for maximum efficiency.

Summary: In order to determine how to minimize execution time the following must be considered:

- 1) Should the test be reversed?
- 2) Should one of the branch points follow the IF?
- 3) If the answer to 2) is yes, which branch point should be chosen?
- 4) Look at the machine language coding actually produced and discover whether the compiler has "outsmarted" you!

Method 4.

LOGICAL IF

The general form of this IF allows several logical expressions to appear in what looks like one great test but the programmer should bear carefully in mind that the order in which the expressions appear may be quite important:

EXAMPLE: IF(A.GT.B.OR.C.LT.D) GO TO 100

This statement implies that if either $A > B$ or $C < D$ the branch to statement 100 should be performed. It is a bit simpler for discussion purposes to break this IF into two IF statements performing exactly the same logical function:

IF(A.GT.B) GO TO 100

IF(C.LT.D) GO TO 100

This is essentially the way in which most compilers will break up the original IF before translating to machine language. There is no strict Fortran rule that the tests are performed in order from left to right so some compilers, in the translating process, could just as well reverse the order:

IF(C.LT.D) GO TO 100

IF(A.GT.B) GO TO 100

Naturally, if the result of one of the two tests, say $A > B$, is expected to be true a large percentage of the time, it behooves the programmer to have this test performed first. Unless the characteristics of the compiler are known (a process of compiling a few sample IF's and looking at the machine language result), it is generally wiser to break complicated IF's into their simpler form to control the order of testing.

For example:

```
IF(A.GT.B.AND.C.LT.D.OR.I.GE.O) GO TO 100
```

This is, logically, really three IF statements

```
IF(A.LE.B) GO TO 1
```

```
IF(C.LT.D) GO TO 100
```

```
1 IF(I.GE.O) GO TO 100
```

Obviously, if I is going to be zero or positive a large percentage of the time the test as written could take on the order of three times as long as it needs to take in actual situations. While the simplicity of stringing the various logical expressions has some advantage in instantaneous recognition of what the code is doing, the application of a little knowledge concerning the probable outcome of the tests could provide overriding advantages in the reduction of execution time.

The remarks made in the discussion of Method 3 concerning either simplifying or removing IF tests from innermost DO loops apply exactly the same as for Method 4.

One question which a programmer should ask is:

How do the following sets of coding compare?

```
a) IF(A-B) 1, 100, 100
```

```
1 CONTINUE
```

```
b) IF(A.GE.B) GO TO 100
```

```
1 CONTINUE
```

In most cases, the resulting code would be comparable but a look at a few examples of the compiler output would point out any special cases where a time advantage could be achieved.

One detail concerns either type of IF statement imbedded inside nested DO loops: the LOAD and TEST will take 1.1 μ sec and the LOAD and SUB will take 1.4 μ sec. The following example illustrates possible gains:

Example 6

```

BEFORE      DO 1 I=1,100
             DO 1 J=1,100
             DO 1 K=1,100
             IF(FLAG-1.) 10,20,10
10  CONTINUE
           - FORTRAN STATEMENTS -
           GO TO 1
20  CONTINUE
           - FORTRAN STATEMENTS -
1   CONTINUE

```

AFTER: FLAGA=FLAG-1.

Example 6A DO 1 I=1,100

```

                     DO 1 J=1,100
                     DO 1 K=1,100
                     IF (FLAGA) 10, 20, 10
                     etc.

```

In fact, depending on the complexity of the loop and on the relative amount of time spent in performing the test as contrasted to the remainder of the calculations in the innermost loop, it is entirely possible that the following can be much faster:

```

Example 6B      IF(FLAG-1.) 100, 200, 100

100 DO 101 I=1,100
               DO 101 J=1,100
               DO 101 K=1,100

10  CONTINUE
   - FORTRAN STATEMENTS -

101 CONTINUE
   GO TO 7

200 DO 201 I=1,100
               DO 201 J=1,100
               DO 201 K=1,100

20  CONTINUE
   - FORTRAN STATEMENTS -

201 CONTINUE

7  CONTINUE

```

Just as constant sub-expressions in arithmetic replacement statements should be removed from inner loops, so also should IF expressions be simplified.

Example 7 DO 1 I=1,10
 DO 1 J=1,20
 Z = A(I,J)+(I,J)
 DO 1 K=1,100
 IF(Z-X(I,J,K)+1.) 2, 3, 4
 etc.

should be rewritten as:

DO 1 I=1,10
 DO 1 J=1,20
 Z = A(I,J)+B(I,J)
 TESTZ = Z + 1.
 DO 1 K=1,100
 IF(TESTZ-X(I,J,K)) 2, 3, 4
 etc.

FINAL DISCLAIMER

All timings given are approximate and depend on various conditions and assumptions as to average use of the computer. The memory timings used here are from IBM SYSTEM/360 MODEL 75 FUNCTIONAL CHARACTERISTICS, Form A22-6889-0. Repeated attempts to obtain more detailed information from IBM concerning the assumptions used have met with failure. No timing given here may be taken as correct but the general considerations are correct.

INTERNAL DISTRIBUTION

- 1-2. Central Research Library
- 3. Document Reference Section
- 4-8. Laboratory Records
- 9. Laboratory Records - Record Copy
- 10-24. Division of Technical Information Extension
- 25. Laboratory and University Div., CRO
- 26. ORNL Patent Office
- 27-76. J. G. Sullivan
- 77. V. R. Cain
- 78. W. B. Gardner
- 79. A. A. Brooks
- 80-730. ORCID List

EXTERNAL DISTRIBUTION

- 731. J. W. Givens, Jr. - Applied Mathematics Division
Argonne National Laboratory, 9700 South Cass Avenue
Argonne, Illinois
- 732. R. P. Leinius - University of Wisconsin Computing Center
1210 W. Dayton Street, Madison Wisconsin
- 733. J. A. Thompson - Control Data Corporation
4201 N. Lexington Avenue, St. Paul, Minnesota
- 734. W. F. Miller - Stanford Linear Accelerator Center
P. O. Box 4349, Stanford, California