CGTM # 47
July 1968
John R. Ehrman

Some Notes on Branching Efficiency in Fortran

This short note is intended as a supplement to SLAC Computer User Note No. 19, which discusses branching problems in detail.

There are several possible conditions that may arise in the use of various branching and control statements that the Fortran user should be aware of; some of these are inherent in the structure of System/360 while others are part of the Fortran language itself.

I. Arithmetic IF statements with fixed-point overflow

Consider the following code segment:

```
      K=0
      IF(K)1,2,1          dummy statement to use statement #1
2     M = 2 ** 30
      IF( M + M ) 3, 4, 5
1     CONTINUE
```

Because the sum in the second IF statement generates a fixed-point overflow, the usual method of branching on condition codes 1, 0, and 2 for negative, zero, and positive results respectively, will not work correctly, because the condition code has been set to 3 by the addition. This means that control will reach statement 1 rather than any of the desired statements. It is not clear what should be done; some of the alternatives are:

(1) to trap all fixed-point overflows by setting the appropriate mask bits in the Program Status Word,

(2) to always branch to one or another of the labels given after the IF statement,

(3) to ignore the condition code setting of the calculation inside the IF statement and perform an additional "Load and Test Register" instruction which will sense the contents of the register as though it had not overflowed,

(4) to follow any IF statement with a fixed-point expression by a branch to an error routine.

Some possible objections to these schemes are, in the same order as above:

(1) the degradation in performance would be high, since each fixed-point overflow would have to be examined; furthermore, the routines of the Fortran Library would have to be recoded almost in their entirety, since many of them make use of instructions which generate such overflows and assume that they are masked off;

(2) there is no easy way to determine which branch is the best to take, and if it is desired to obtain some sort of optimized form of the branches then it will be difficult to take the desired branch in case of overflow;

(3) this leads to peculiar results since the sign of the overflowed result is almost always the opposite of the sign of the two original operands (as in the example above, where a test for sign would branch to statement number 3);

(4) this would require extra code in each such IF statement, and it isn't clear what the program should do other than inform the programmer that he fell through the IF.

IBM is well aware of the problem, and it has been under discussion for about a year with SHARE. If you have any suggestions, please let me know.

II. Computed GO TOs with invalid branch index

In a computed GO TO statement such as

```
GO TO (5,15,27),K
```

it is possible that the branch index K has a value that is not 1, 2, or 3. In such a situation it is not clear as to what action should be taken; in the System/360 Fortrans the branch is ignored, and control passes to the statement following the Computed GO TO. This has several implications for the programmer: the logical flow of his program may not be what he expects, and the program must assume that there is a legal path from the GO TO to the following statement in the program, and adjust any optimization information accordingly. Some possible solutions are

(1) to branch to a location relative to the correct ones as determined by the true value of the branch index (as was done in 7090 Fortran),

(2) branch to the first statement number if the index is less than or equal to 1, or to the last if the index is greater than or equal to the largest legal value, or

(3) to branch to an error routine in the system whenever such an erroneous branch is attempted.

Some possible objections to these solutions are:

(1) this is an almost certain way to wind up somewhere in the sticks, and is even worse on the 360 since instructions do not fall on regular boundaries;

(2) the logical flow of the program is not correctly related to the values it has computed; and

(3) extra code will be required for each computed GO TO in the program.

III. Incautious use of the Assigned GO TO

The Assigned GO TO is the fastest way to set up conditional branches in a program, but it is possible to generate errors if the same variable is used both for assigned statement numbers and for arithmetic purposes. For example, in the following program segment the programmer would have to be careful not to fall afoul of the conflicting uses of the variable K:

```
ASSIGN  20  TØ  K
.......
GØ TØ K, (6, 20, 127, 44)
.......
K = 20
.......
```

One further minor point: in the H compiler with OPT=2, the list of statement numbers following the "GO TO K" must not have any missing elements, or the branch may generate incorrect code.

IV. Logical IF statements with compound conditions

The logical IF statement provides a convenient way to test several conditions at once and to take appropriate action depending on the result. For example, assuming that A, B, C... are logical variables or are a shorthand form for some logical expression such as X.GE.Y, then Logical IF statements consisting

of a series of ANDs such as

```
IF (A .AND. B .AND. C .AND.  ...  )  statement
```

should be rewritten to avoid having to calculate all the logical conditions, since it is known that if one of them is false then the entire conditional expression will be false. Thus one should therefore write

```
      IF ( Ā ) GO TO 999
      IF ( B̄ ) GO TO 999
      IF ( C̄ ) GO TO 999
      ......
      statement
999   CONTINUE
```

where by $\bar{A}$ we mean the opposite condition to that expressed by A. Similarly, for a string of ORs such as

```
IF (A .OR. B .OR. C .OR.  ... )  statement
```

one should write instead

```
      IF ( A ) GO TO 999
      IF ( B ) GO TO 999
      IF ( C ) GO TO 999
      ......
      GO TO 777
999   statement
777   CONTINUE
```

More complicated logical expressions can be broken down in much the same manner into single tests; the idea is to avoid having to do unnecessary comparisons and calculations wherever possible. One must assume that the compiler cannot determine which portions of the logical expression need not be recomputed, but that it must compute the entire expression each time it is encountered. Thus, code such as

```
      IF ( A ) do something
      IF ( A ) do something else
      IF ( A ) do still something else
```

will require that the logical expression A be recalculated each time.

V. Logical IF statements with Arithmetic IF trailer

Under certain circumstances it is useful to know that one can write a statement of the form

IF (logical expression) IF (arithmetic expression) a, b, c

This may not generate the best possible code for the given conditions, but is simple and fairly clean.