X-543-71-

OPTIMIZATION TECHNIQUES FOR FORTRAN IV (G AND H) PROGRAMS WRITTEN FOR THE IBM 360 UNDER OS

MARCH 1971

PRICES SUBJECT TO CHANGE

GSFC

GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND

X-543-71-99

OPTIMIZATION TECHNIQUES FOR FORTRAN IV (G AND H) PROGRAMS WRITTEN FOR THE IBM 360 UNDER OS

John L. Dean

Prepared Under Contract NAS5-11799 by
Boole & Babbage, Inc.
Cupertino, California

for Analysis & Programming Branch,
Computation Division, Tracking and
Data Systems Directorate,
Goddard Space Flight Center
Greenbelt, Maryland
National Aeronautics and Space Administration

March 1971

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

CONTENTS

OPTIMIZATION TECHNIQUES FOR FORTRAN IV (G AND H) PROGRAMS WRITTEN FOR THE IBM 360 UNDER OS

INTRODUCTION

The purpose of this paper is to provide a fairly complete list of programming techniques which are available to the programmer for optimizing the execution time of production programs written in FORTRAN IV (G and H) for the IBM 360 under OS.

A number of the techniques presented herein rely on a tradeoff between extra storage and execution time, and it is not suggested that they be used throughout a program. Rather, the programmer should establish the high usage areas of his program and concentrate his optimization efforts within these areas. For existing production programs, the Boole & Babbage PPE (Problem Program Evaluator) product can be used to identify program areas of high utilization.

The first step in any optimization effort should be to recompile the program under FORTRAN H, OPT=2. This will almost always give a very significant performance improvement, averaging about 25-30% savings. 50% and greater savings are not uncommon. The amount will, of course, vary considerably from program to program. As a general rule, the greater the use of loops, and the degree of nesting of loops, the greater the improvement. The reliability of the code generated under OPT=2 is vastly improved with OS/360 release 18.6 and up.

After the program has been compiled under FORTRAN H, OPT=2, then the process of actually changing code can begin. The bulk of execution time of FORTRAN programs can almost always be attributed to a few loops, and primary consideration should be given to these. Depending upon the individual program, the programmer may wish to consult other sections of this paper for more specific suggestions concerning the various types of FORTRAN statements.

DO LOOPS

As a general rule, the execution of DO loops accounts for the bulk of execution time of most FORTRAN programs. For this reason the greatest concentration of any optimization effort should be made in this area. A few areas of importance are:

1. Avoiding Redundant Calculations

Example 1

```
      DO 10 I = 1,30
      D(I) = C(I) * A * B
10    E(I) = E(I) + (A * B) * * 2
```

In the above loop the quantity [A * B] is calculated 60 times, and the quantity [(A * B) * (A * B)] 30 times (90 multiplications). In the entire loop there are 120 multiplications and 30 additions. The loop should have been coded:

```
      AB = A * B
      ABSQR = AB**2
      DO 10 I = 1,30
      D(I) = D(I) * AB
10    E(I) = E(I) + ABSQ
```

Here, there are a total of 32 multiplications and 30 additions, a net savings of 88 multiplications.

Example 2

```
      SUM = 0.D0
      DO 10 I = 1,30
10    SUM = SUM + C(I) * B
```

The unnecessary calculations above are slightly less obvious, but exist nonetheless. Note that there are 30 additions and 30 multiplications performed. A better method would have been:

```
      SUM = 0.D.0
      DO 10 I = 1,30
10    SUM = SUM + C(I)
      SUM = SUM * B
```

Here, the same net result is obtained, but with 30 additions and 1 multiplication. In addition, the final value of SUM will be more accurate

2. Subscripting

Example 1

```
      DO 10 J = 1,6                Method 1
10    KK(3 * J) = J + 3

      K = 0
      DO 11 J = 1,6                Method 2
      K = K + 3
11    KK(K) = J + 3
```

Here, method 1 will execute in approximately 90% of the time required for method 2. The reason for this is that the increased bookkeeping necessary for keeping track of the variable K is greater than the savings in avoiding a multiplication in subscript calculation.

Example 2

```
      DO 10 I = 1,6
      J = I + 1                    Method 1
      K = I + 2
      L = I + 3
      M = I + 4
10    A(J,K) = B(L,M)

      DO 11 I = 1,6                Method 2
11    A(I + 1, I + 2) = B(I + 3, I + 4)
```

In this example, an attempt was made to simplify the coding of the indexes on A and B. However, method 2 will execute in only two thirds of the time of method 1.

Example 3

```
      DO 10 I = 2,80,2             Method 1
10    A(I) = B(I + 2, I + 2)

      DO 11 I = 2,80,2             Method 2
      J = I + 2
11    A(I) = B(J,J)
```

In method 2, an attempt was made to avoid an apparent redundant calculation of the value (I + 2) in method 1. In fact, the compiler recognized the duplicate subscript value, and calculated it only once, thus method 1 is the better technique.

Example 4

```
      DO 10 I = 1,6                    Method 1
      LL(K) = LL(K) + 1
10    C(LL(K)) = D(LL(K)) * E (LL(K))

      LLA = LL(K) + 1                  Method 2
      LLB = LLA + 5
      DO 11 I = LLA, LLB
11    C(I) = D(I) + E(I)
      LL(K) = LL(K) + 6
```

This example illustrates how subscript calculations may be eliminated from a loop via careful choice of loop parameters. In method 2 the variable LL(K) is incremented only once. In addition, the variable I serves double duty as the DO variable, and as an index. Method 2 will execute in approximately 60% of the time needed for method 1.

Example 5

```
      DIMENSION X(10, 10)
       .
       .
       .
      DO 20 I = 1,10                   Method 1
      DO 20 J = 1,10
20    X(I, J) = 0.

      DO 21 I = 1,100                  Method 2
21    X(I, 1) = 0.
```

In spite of the fact that X is a two dimensional array it may be logically treated as though it had only one dimension as shown in method 2. In this manner, a great deal of the time spent in calculating indexes within a loop can be avoided. In the example shown, method 2 will execute in approximately two thirds of the time of method 1.

3. Loop Elimination

In high usage areas of a FORTRAN program small loops should be written out in-line. This is particularly true of the innermost loop(s) of a series of nested loops. FORTRAN loops are extremely expensive, with overhead occurring both in controlling the loop and in calculating indexes for variables arranged in arrays which are used within the loop.

Example 1

```
      DO 10 J = 1,6
10    KK(3 * J) = J + 3
```

The above loop will cause 92 machine language instructions to be executed. If we write the loop in-line as

```
KK(3) = 4
KK(6) = 5
KK(9) = 6
KK(12) = 7
KK(15) = 8
KK(18) = 9
```

Only 13 machine language instructions will be executed, a savings of approximately 85%.

Example 2

```
      SUM = 0.
      DO 10 I = 1,6
10    SUM = SUM + D(I) * E(I)
```

This is a very common technique for accumulating the product of two vectors. The entire loop could have been written out in one statement as

```
SUM = D(1) * E(1) + D(2) * E(2) + ... + D(6) * E(6)
```

with the single statement taking approximately one third as much execution time as the loop.

Example 3

In the following pair of nested loops, the outer rather than the inner loop is a known value

```
      DO 10 I = 1,3
      DO 10 J = 1,N
10    X(I, J) = Y(I, J) + Z(J, I)
```

We can nonetheless remove one level of indexing and obtain a significant performance gain in the following manner

```
      DO 10 J = 1,N
      X(1, J) = Y(1, J) + Z(J, 1)
      X(2, J) = Y(2, J) + Z(J, 2)
10    X(3, J) = Y(3, J) + Z(J, 3)
```

Example 4

Suppose that in the following loop, the index variable can indeed vary from run to run

```
      DO 10 I = 1,N
10    A(I) = B(I)
```

On the face of it, nothing can be done to improve this loop. If it were known, however, that most of the time the value of N was 6, then the following can be done.

```
      IF(N.EQ.6) GO TO 11
      DO 10 I = 1,N
10    A(I) = B(I)
      GO TO 12
11    A(1) = B(1)
      A(2) = B(2)
      A(3) = B(3)
      A(4) = B(4)
      A(5) = B(5)
      A(6) = B(6)
12    CONTINUE
```

It is not suggested that this technique be used unless the loop in question is known to have high utilization.

Example 4a

Taking the loop in Example 4, if it were known that the value of N ranged from 5 upwards, the following could be done

```
      A(1) = B(1)
      A(2) = B(2)
      A(3) = B(3)
      A(4) = B(4)
      A(5) = B(5)
      IF(N.EQ.5) GO TO 11
      DO 10 I = 6,N
10    A(I) = B(I)
11    CONTINUE
```

FUNCTIONS AND SUBROUTINES

Functions and subroutines are used in a program to either reduce storage or for programming convenience or both. In general, the concept of modularity is a good one, however it can be misused, particularly where small subprograms are involved. For such small subprograms, it is quite possible for the overhead involved in passing arguments, saving registers, etc. to be greater than the actual work accomplished. In addition, such functions or subroutines must be coded as general purpose routines, making optimization of them difficult.

As a general statement, then, high usage small subprograms should be coded in-line in the calling routine. As a corollary to this, routines which are known to have high utilization and call a number of small subprograms, should have such calls coded in-line.

Example 1

The following is excerpted from an actual production program.

```
      TERMS(1) = X
      TERMS(2) = -OMU * TEM0/R1C
      TERMS(3) = -MU * TEM/R2C
      TERMS(4) = 2.D0 * YDOT
      XDDOT = SUM(TERMS, 4)
        .
        .
        .
      DOUBLE PRECISION FUNCTION SUM(CKTRM, NTRMS)
      DOUBLE PRECISION CKTRM, SUMP, SUMN
      SUMN = 0.D0
      SUMP = 0.D0
      DO 15 I = 1,NTRMS
      IF(CKTRM(I)) 5,10,10
    5 SUMN = SUMN + CKTRM(I)
      GO TO 15
   10 SUMP = SUMP + CKTRM(I)
   15 CONTINUE
      SUM = SUMP + SUMN
      RETURN
      END
```

The purpose of function SUM is to simply add the desired number of terms (NTRMS) of the supplied matrix (CKTRM). It was determined

through using the Boole & Babbage PPE package that nearly 40% of the execution time of the whole program was taken up in function SUM. There were only two other invocations of SUM in the program, both similar to that shown above. The ENTIRE 40% was saved by recoding as follows:

```
      XDOT = X - OMU * TEM0/R1C - MU * TEM/R2C + 2.D0
          * YDOT
```

Example 2

```
      DIMENSION X(6), Y(6), Z(6)
      CALL SUBROUTINE CROSS(X, Y, Z, 6)
        .
        .
        .
      SUBROUTINE CROSS (A, B, C, N)
      DIMENSION A(1), B(1), C(1)
      DO 10 I = 1,N
10    C(I) = A(I) * B(I)
      RETURN
      END
```

In the above, subroutine CROSS is used as a convenient means of performing cross multiplication. Note, however, that since the loop is controlled by an unknown variable, the subroutine cannot be made more efficient. If it could be determined that the particular call to CROSS was performed many times during the course of a run, then a very significant performance gain can be had by recoding the call as:

```
      DIMENSION X(6), Y(6), Z(6)
      Z(1) = X(1) * Y(1)
      Z(2) = X(2) * Y(2)
      Z(3) = X(3) * Y(3)
      Z(4) = X(4) * Y(4)
      Z(5) = X(5) * Y(5)
      Z(6) = X(6) * Y(6)
```

thus avoiding both the overhead of a subroutine call and the loop.

Example 3

Multiple entry subprograms can become very expensive if similar but different functions are performed for each entry point and if common

code is executed for each entry. The following routine was taken from a widely used production program.

```
      DOUBLE PRECISION FUNCTION DOT(X, Y)
      IMPLICIT REAL*8(A-H,O-Z,$)
      DIMENSION   A(6),   SUM(3)
      DIMENSION X(3), Y(3), Z(3), W(3)
      DATA OVT/192./
10    II = 1
      J2 = 4
      GO TO 32
      ENTRY ADOT (X,Y)
      C = 57.2957795131DØ
      GO TO 31
      ENTRY ADOTR (X,Y)
30    C = 1.DØ
31    II = 2
      J2 = 1
32    DO 33 I = 1,3
      A(I) = X(I)
33    A(I + 3) = Y(I)
      GO TO 101
34    I = I + 1
      GO TO (35,35,36,37) ,I
35    J1 = 1
      GO TO 102
36    J2 = 4
      GO TO 102
37    SCL = DMAX1(SUM(1),SUM(3))
      IF (SCL.LE.OVT) GO TO 39
      DO 38 I = 1,3
38    SUM(I) = SUM(I)/SCL
39    SUM(I) = C*ARKTNS(180,SUM(2),DSQRT(SUM(1)*SUM(3)-
          SUM(2)*SUM(2)))
      GO TO 999
      ENTRY FNORM (Z)
40    II = 3
      GO TO 51
      ENTRY VNORM (Z,W)
50    II = 4
51    J2 = 1
      DO 52 I = 1,3
52    A(I) = Z(I)
      GO TO 101
```

```
 53   SUM(1) = DSQRT(SUM(1))
      IF (II.LE.3) GO TO 999
      DO 54 I = 1,3
 54   W(I) = A(I)/SUM(1)
      GO TO 999

1Ø1   I = 1
      J1 = 1
1Ø2   SUM(I) = Ø.DØ
      DO 1Ø3 J = 1,3
      SUM(I) = SUM(I) + A(J1)*A(J2)
      J1 = J1 + 1
1Ø3   J2 = J2 + 1
      GO TO (999,34,53,53) ,II
999   DOT = SUM(1)
      RETURN
      END
```

DOT was the most heavily used entry point in this routine. The following code was executed for each entry to DOT:

```
 10   II = 1
      J2 = 4
      GO TO 32
 32   DO 33 I = 1,3
      A(I) = X(I)
 33   A(I + 3) = Y(I)
      GO TO 101
101   I = 1
      J1 = 1
102   SUM(I) = 0.D0
      DO 103 J = 1,3
      SUM(I) = SUM(I) + A(J1) * A(J2)
      J1 = J1 + 1
103   J2 = J2 + 1
      GO TO (999,34,53,53), II
999   DOT = SUM(1)
      RETURN
```

When this is all boiled down to basics we find that the above can simply be replaced by

```
DOT = X(1) * Y(1) + X(2) * Y(2) + X(3) * Y(3)
```

By coding a number of references to DOT in-line in the calling routines, a very significant savings in program run time was achieved.

FORTRAN I/O

The FORTRAN language was designed primarily for the purpose of executing computational algorithms, rather than for performing input/output operations. Accordingly, FORTRAN I/O is grossly inefficient. The execution time required by the FORTRAN I/O routines very often accounts for somewhere between 10% to 50% of total CPU time for FORTRAN jobs!

There are four ways by which the I/O charge for a FORTRAN job can be reduced. The first is to simply limit the amount of I/O being performed. If this is unacceptable or impossible, then there are three programming techniques that can be used which, if applicable, can vastly decrease I/O overhead.

1. Blocking

Blocking of data sets will reduce the number of actual I/O operations which must be made to read or write a given amount of data. The CPU time saved by this is quite considerable, and is worth the extra storage required for larger buffers (this can be mitigated by using only one buffer).

Example 1

The following loop might be used to output a large array onto an intermediate work file in 80 byte pieces.

```
      REAL * 8 A(1000)
        .
        .
        .
      K = 1
      L = 10
      DO 10 J = 1,100
      WRITE (8) (A(I), I = K,L)
      K = K + 10
10    L = L + 10
```

If a DD card of the following form were used at execution time, then 100 actual write operations would be needed to output the matrix onto file 8.

```
//FT08F001 DD ... DCB = (RECFM = VBS, LRECL = 84,
    BLKSIZE = 88) ...
```

By changing the DD statement to

```
//FT08F001 DD ... DCB = (RECFM = VBS, LRECL = 84,
    BLKSIZE = 4204) ...
```

the number of writes will be decreased from 100 to 2. Note: for direct access devices the blocksize should not exceed a full track (e.g., 7294 bytes for a 2314).

2. Unformatted I/O

Very often large amounts of data are output onto intermediate storage devices such as tapes or disk for later processing, either by the same program or by another program. If such data are written and later read under Format control then much CPU time will be unnecessarily wasted. When a list of items is being read/written under Format control, each item in the list (each element of an array is considered an individual item) must be passed to special programs in the FORTRAN I/O package for the necessary conversion. If this work can be bypassed, then much will be gained.

Example 2

```
      REAL * 4 X(100)

      WRITE(8,111) X
111   FORMAT(100F8.3)
       .
       .
       .
      READ(8,111) X
```

Suppose that it were desired to write out the array X onto a temporary work file and then later read it back in for further processing as shown above. The DD statement for file 8 might look like this.

```
//FT08F001 DD ... DCB = (RECFM = FB, LRECL = 800,
    BLKSIZE = 800)
```

During the course of reading and writing this data, the FORTRAN formatting routines would be entered a total of 200 times.

If the data were to be read/written in binary form, then the program statements would look like

```
WRITE (8) X
  .
  .
  .
READ (8) X
```

and the DD card for file 8 might be

```
//FT08F001 DD ... DCB = (RECFM = VBS, LRECL = 404,
     BLKSIZE = 408)
```

This will provide better performance in several ways. The greatest benefit being the CPU savings due to eliminating formatting. In addition there will be a storage savings because smaller buffers are needed for the unformatted I/O. This will in turn give some additional CPU savings because less data must be transmitted. A final item is that when the formatted items are read back into storage they will not be as accurate as when they were written. This is because they would have had their least significant digits truncated to 3 decimal place accuracy as part of the formatting process. This latter difficulty can be avoided by using a sufficiently large field, however this will increase both storage and processing time.

3. Array Reads/Writes

Example 3

```
      DIMENSION A(100)

      WRITE(6,111) (A(I), I = 1,100)
111   FORMAT(10(1X,F10.2))              Method 1

      WRITE(6,111) A                    Method 2
```

Both write statements above will accomplish the same result, namely to print array A. Method 2, however, is much faster than method 1. The difference occurs in the manner in which list items in FORTRAN I/O statements are handled. For each item in the list, a call is made from the program to the FORTRAN I/O interface routine (IBCOM =). In method 1, there will be 100 subroutine calls (with associated overhead)

necessary to print array A. Note that a DO loop is implied [(A(I), I = 1,100)] within the Write statement. This loop must be controlled from the program itself, as the loop implies that the programmer wishes to control the order in which the array elements are to be printed.

In method 2 the array A is treated as only one list item despite the fact that it has 100 elements. This can be done because the Write statement implies that the entire array is to be written. In this instance the looping from element to element through the array will be controlled within the FORTRAN I/O interface routine itself, thereby avoiding 99 of the 100 calls to this routine needed by method 1.

Example 4.

```
DIMENSION A(1000)
  .
  .
  .
WRITE(8) (A(I), I = 701,800)
```

In the write statement above, only a segment of A is to be written, which would seem to indicate that the technique shown in example 3 would not apply. By the use of an EQUIVALENCE statement, however, the savings may still be had.

```
DIMENSION A(1000), B(100)
EQUIVALENCE (A(701), B(1))
  .
  .
  .
WRITE(8) B
```

Example 5

An equivalence statement can also be used to reduce I/O processing time for a list of individual items. For instance, the Write statement

```
WRITE(4) A,B,C,D,E,F,G,X,Y,Z
```

could be replaced by

```
DIMENSION W(10)
EQUIVALENCE (W(1),A), (W(2),B), ... (W(10),Z)
```

```
      .
      .
      .
      WRITE(4) W
```

Example 6

```
      WRITE(6,111) (A(I), I = J,K)
111   FORMAT(10(1X,F8.3))
```

Here, a segment of unknown size is being printed from array A. The many calls to the I/O interface routine can still be avoided through use of a subroutine, i.e.,

```
      NUMB = K-J + 1
      CALL OUTPUT (A(J), NUMB)
       .
       .
       .
      SUBROUTINE OUTPUT (ARRAY,N)
      DIMENSION ARRAY(N)
      WRITE(6,111) ARRAY
111   FORMAT(10(1X,F8.3))
```

Example 6a

The output subroutine could have been made fully general purpose in the following manner

```
      DIMENSION NFORM(4)
      DATA NFORM/'(10(1X,F8.3))'/
       .
       .
       .
      NUMB = K-J + 1
      IUNIT = 6
      CALL OUTPUT (A(J), NUMB, IUNIT, NFORM)
       .
       .
       .
      SUBROUTINE OUTPUT (ARRAY, N, IUNIT, NFORM)
      DIMENSION ARRAY (N), NFORM (1)
      WRITE (IUNIT, NFORM) ARRAY
```

FORTRAN MATHEMATICAL LIBRARY SUBPROGRAMS

Functions such as SIN, COS, SQRT, etc. are widely used in FORTRAN programs. These functions are very convenient, but they are also very expensive. Most of them require a series expansion of some kind, error checking, and occasionally calls to other library routines. Accordingly, unnecessary use of such functions should be avoided.

Example 1

Quite often multiple calls to library functions are made as a programming convenience, such as

```
A = SIN(THETA) + 1.
B = COS(THETA) + D
C = 2. * SIN(THETA) + 3. * COS(THETA)
```

This should be recoded as:

```
THSIN = SIN(THETA)
THCOS = COS(THETA)
A = THSIN + 1.
B = THCOS + D
C = THSIN + THSIN + 3. * THCOS
```

Example 2

```
      DO 20 I = 1,10
      DO 20 J = 1,10
      A(I, J) = SIN(C(I))
20    B(I, J) = COS(C(I))
```

In the above loop the SIN and COS routines are each used 100 times. If, however, we were to recode the loop in the following manner, this would be reduced to 10 each.

```
      DO 20 I = 1,10
      SINC = SIN(C(I))
      COSC = COS(C(I))
      DO 20 J = 1,10
      A(I, J) = SINC
20    B(I, J) = COSC
```

Example 3

Use of trigonometric identities can reduce running times

```
A = SIN(THETA)**2        Method 1
B = COS(THETA)**2

A = SIN(THETA)**2        Method 2
B = 1.-A
```

Obviously method 2 is the faster. It takes advantage of the trigonometric identity:

$$SIN^2(x) + COS^2(x) = 1$$

EXPONENTIATION

Exponentials can be expensive and should be avoided if possible. Some forms of exponentiation are more expensive than others. For instance:

```
A = B * * 4                 Method 1

A = B * * 4.0               Method 2

I = 4
A = B * * I                 Method 3
```

The three techniques above are equivalent in terms of the resultant value of A, however method 1 is accomplished via in-line code generated by the compiler, whereas methods 2 and 3 require calls to FORTRAN Library Routines in order for the value of A to be calculated. In general, raising a variable to an integer constant power is preferable to any other technique including serial multiplication.

```
A = B * B * B * B           Method 4
```

Method 1 is still the fastest technique. Because it knows the exact power to be taken, the compiler can optimize the necessary calculations. The value of A will be calculated in method 1 via two register-to-register multiplications, while method 4 will require three register-to-storage multiplications, thus method 1 is not only faster, but will involve less round-off error.

Example 1

```
T1 = A * * 3 + B * * 3
T2 = C * A * * 3 + 2.* B * * 3
T3 = D * A * * 3 * B * * 3
```

In the above sequence of statements the values A * * 3 and B * * 3 are each calculated three times. A better method would be

```
ACUBE = A * * 3
BCUBE = B * * 3
T1 = ACUBE + BCUBE
T2 = C * ACUBE + 2.* BCUBE
T3 = D * ACUBE * BCUBE
```

Example 2

```
X(I) = C * Y * * I + D * Y * *(I + 1)
```

In the above statement it will be necessary to branch twice to a FORTRAN Library Routine to calculate powers of Y. One of these calls can be avoided by factoring as follows:

```
X(I) = (C + D * Y) * Y * * I
```

Example 3

Suppose the statement from example 2 were to appear in a DO loop such as

```
      DO 10 I = 1,K
10    X(I) = C * Y * * I + D * Y * *(I + 1)
```

In this instance, all calls to the exponential routines can be eliminated as follows:

```
      TEMP = 1.0
      DO 10 I = 1,K
      TEMP = TEMP * Y
10    X(I) = (C + D * Y) * TEMP
```

Example 4

Constants raised to integer powers are often used in loops for coding or decoding purposes. For instance

```
      X = 0.
      DO 10 I = 1,N
10    X = X + A(I) * 10. * * I
```

Each time the loop is executed, the constant 10. must be raised to an integer power N times, with a call to a library routine required for each. If ascending powers of 10 were placed in a table, then the loop could become

```
      X = 0.
      DO 10 I = 1,N
10    X = X + A(I) * TABLE(I)
```

providing a very drastic decrease in execution time.

MIXED MODE EXPRESSIONS

The use of mixed mode expressions is a very costly business. The process of conversion of numbers from one form of representation to another is quite expensive, and it is worth some programming effort to avoid.

Example 1

If it is absolutely necessary for mixed mode expressions to be used, then calculations should be grouped by type to the maximum extent possible.

```
A = J + AJ + K + AK + L + AL + J*K + AJ*AK + J*L + AJ*AL
    + J*AL*K
```

Method 1

```
A = (J + K + L + J*K + J*L) + (AJ + AK + AL + AJ*AK + AJ*AL
    + J*K*AL)
```

Method 2

Method 2 will execute in one half the time of method 1.

Example 2

A quite common technique is to use the DO loop variable both as a counter and as a variable within an expression, such as

```
      DO 10 I = 1,N
      A(I) = C * FLOAT(I)
10    B(I) = D + FLOAT(2*I)
```

Each pass through the loop will involve two conversions from integer to real. A much more efficient method would be:

```
      XI = 0.
      DO 10 I = 1,N
      XI = XI + 1.
      A(I) = C*XI
10    B(I) = D + XI + XI
```

thus avoiding the two conversions (and also a multiplication).

COMMON AREAS

Common areas are used either to provide working space which can be shared by subprograms (reducing total program size) or to reduce the length of calling sequences between subprograms. This latter use can reduce program running times if many calls are involved.

Example 1

```
CALL SUB1(A, B, C, D)                Method 1
  .
  .
  .
SUBROUTINE SUB(A, B, C, D)
D = A + B + C
RETURN
END

COMMON A, B, C, D                    Method 2
CALL SUB2
  .
  .
  .
SUBROUTINE SUB2
COMMON A, B, C, D
D = A + B + C
RETURN
END
```

Subroutine SUB2 in method 2 will execute in roughly half the time of SUB1. The difference here is the time necessary to decode the calling sequence in SUB1, plus extra handling to ensure that the answer is stored back in the proper area of the calling program. This, of course, is a fixed amount of work, and the percentage savings would be much less in a larger subroutine (but a savings nonetheless).

There are ways in which Common areas can be misused, all involving placing variables into Common in a nonoptimal manner.

Example 2

```
REAL * 8 A(1000), B(10)
COMMON A, B
```

The Common area as defined above is perfectly legitimate. Since the array A is 8000 bytes in length, any statement referring to both A and B will require the use of two base registers. This is because the beginning location of B is displaced by more than 4096 bytes from that of A. By switching the variances in Common to be

```
REAL * 8 A(1000), B(10)
COMMON B, A
```

only one base register will be needed. This can have an effect on program execution time, particularly if FORTRAN H is being used, with an optimization level of 2.

Example 3

```
COMMON/ONE/A, B, C
COMMON/TWO/D, E, F
COMMON/THREE/X, Y, Z
```

Each independent Common area will require one base register, for a total of three registers which could possibly be tied up at one time. If possible, multiple Named Common areas should be combined, such as

```
COMMON/ALL/A, B, C, D, E, F, X, Y, Z
```

Example 4

It is possible to define Common areas in such a way that some variables may be improperly aligned.

```
INTEGER * 4 I, J, K
REAL * 8 X
COMMON I, J, K, X
```

In the Common area above, the Double Precision variable X is not aligned on a double word boundary relative to the beginning of the Common area. During execution, any reference to X will cause a specification error.

At this point two things can happen. If boundary alignment error recovery is not SYSGENED into the system under which the program is executing, the job will abend, and it can be assumed that the problem will be found and corrected. If it is included, then the specification

error will be ignored, and a special boundary alignment routine will be invoked which will allow execution to continue. This is obviously quite expensive, but unfortunately will be largely transparent to the programmer, thus it is important that proper alignment is achieved in the first place in order to avoid degradation during program execution.

(Note: the recovery procedure is not SYSGENED in T&DS 360 systems.)

IF STATEMENTS

A logical IF statement will execute in equal or less time than an equivalent arithmetic IF.

Example 1

```
        IF(A.GT.B) GO TO 20          Method 1

        IF(A-B) 10, 10, 20           Method 2
10      CONTINUE
```

Method 1 is preferred. It not only may be faster, but is also more understandable to someone examining a FORTRAN program.

If arithmetic IF statements are used, the IF statement should be immediately followed by one of the statements which can be branched to

Example 2

```
        IF(A-B) 10, 10, 20           Method 1
 5      CONTINUE

        IF(A-B) 10, 10, 20           Method 2
10      CONTINUE
```

Method 2 should be used. The difference here is that in method 1 a branch to statement 10 will cause 2 machine language instructions to be executed, while in method 2 none will be necessary.

In using compound logical IF statements, execution time can be saved by proper ordering of logical expressions within the IF statement.

Example 3

```
        DO 10 I = 1,100
        IF(I.LT.9 .OR. I.GT.50) GO TO 10
          .
          .
          .
10      CONTINUE
```

The IF statement in the loop above is compiled exactly as though it were written:

```
      IF(I.LT.9) GO TO 10
      IF(I.GT.50) GO TO 10
```

During execution, the first statement will be executed 100 times, and the second statement 92 times for a total of 192. Since I is greater than 50 more often than it is less than 9, their order should be reversed to become

```
      IF(I.GT.50) GO TO 10
      IF(I.LT.9) GO TO 10
```

In this case, the first statement will be executed 100 times, but now the second will only be executed 50 times for a total of 150 (a 22% saving). Accordingly the original IF statement should be written as

```
      IF(I.GT.50.OR.I.LT.9) GO TO 10
```

Similar savings may be achieved where .AND. is used in logical IF statements, but here we will try to arrange expressions in such a manner that those most likely to be not true will be evaluated first.

Example 4

```
      DO 10 I = 1,100
      IF(I.GT.9 .AND. I.LT.50) GO TO 10
        .
        .
        .
10    CONTINUE
```

Here, the IF statement effectively expands to:

```
      IF(I.LE.9) GO TO 20
      IF(I.LT.50) GO TO 10
20    CONTINUE
```

Since I is less likely to be less than 10 than greater than 49, the IF statement should appear as

```
      IF(I.LT.50 .AND. I.GT.9) GO TO 10
```

Both .AND. and .OR. should not appear in the same logical IF statement. If they do, the entire expression will be evaluated regardless of the ordering of the statement.

GO TO STATEMENTS

If 4 or more statement labels are used in a Computed GO TO statement it will usually be faster than a series of logical IF statements.

Example 1

```
IF(I.EQ.1) GO TO 10                    Method 1
IF(I.EQ.2) GO TO 20
IF(I.EQ.3) GO TO 30
IF(I.EQ.4) GO TO 40

GO TO (10, 20, 30, 40), I              Method 2
```

Given equal probability of I assuming any value from 1 to 4, method 2 is faster than method 1.

Example 2

```
IF(I-2) 10, 20, 30                     Method 1

GO TO (10, 20, 30), I                  Method 2
```

Here, method 2, using the Computed GO TO will be slower.

The assigned GO TO is the fastest possible conditional branch.

Example 3

```
ASSIGN 10 to I                         Method 1
  .
  .
  .
GO TO I, (10, 20, 30, 40, 50)

I = 1                                  Method 2
  .
  .
  .
IF(I.EQ.1) GO TO 10
```

```
I = 1                                    Method 3
  .
  .
  .
GO TO (10, 20, 30, 40, 50), I
```

Of the three possible means of branching to statement number 10 shown above, method 1, using the Assigned GO TO is twice as fast as either of the alternatives.

MISCELLANEOUS

1. Data Statement

Very often in subprograms which perform a series of calculations, there will be a number of statements at the beginning of the subprogram which initialize constants to be used later in the routine.

```
C1 = 1.0
C2 = 0.5
C3 = 25.7
```

If these variables are never changed during execution of the subprogram, then a DATA initialization statement should be used in order to avoid this unnecessary work for each call.

```
DATA C1, C2, C3/1.0, 0.5, 25.7/
```

2. Factoring

Example 1

Consider the statement

```
A = A + B *(D * E + F) - C *(F + D * E)
```

By rewriting this as

```
A = A + (B-C) * (D * E + F)
```

two multiplications and one addition are saved.

3. Redundant Calculations

Program execution time will be reduced if multiple executions of a given expression can be recognized and eliminated.

Example 1

```
A = B + C + D * E + X * * 2 - F/G
.
.
.
```

```
S = S + D * E - Y + B - F/G + D * E
  .
  .
  .
T = T - F/G + B + 2 + D * E
```

The three statements above all contain the common expression

```
(B + D * E - F/G)
```

and could be rewritten as

```
TEMP = B + D * E - F/G
A = TEMP + C + X * * 2
  .
  .
  .
S = S + TEMP - Y
  .
  .
  .
T = T + TEMP + 2
```

4. Temporary Variables

Very often evaluation of a large expression will be broken down into several smaller steps, with intermediate results being stored in temporary variables. After the sub-calculations are completed the temporary variables are summed to obtain the final result. This technique is presumably used in order to make keypunching easier or to make the program more understandable, or both. While convenient in these respects, it does slow down program execution and should be avoided.

Example 1

```
T1 = A * * 2 + B * * 2 + C * * 2
T2 = W + X + Y - W * X * Y
T3 = (D + E - F) * * 3 * (D - E)
ANSWER = T1 + T2 + T3
```

This series of statements should be combined into one statement. Note that this can be done while preserving readability and/or keypunchability.

```
      ANSWER = A * * 2 + B * * 2 + C * * 2
1                    + W + X + Y - W * X * Y
2                    + (D + E - F) * * 3 * (D - E)
```

APPENDIX A

BIT AND BYTE MANIPULATION

Neither bit nor byte manipulation are considered to be an integral part of IBM FORTRAN (i.e., they are not supported). Nonetheless, facilities do exist which allow these kinds of operations to be done in a fast and convenient fashion. Since they are not supported, their description is confined to an Appendix. They are included in this paper, because in certain special situations their use can drastically reduce program running times.

BIT MANIPULATION (FORTRAN H ONLY)

Although there is no documentation in either the FORTRAN IV Language Manual or Programmer's Guide as to their existence, there are a class of compiler generated in-line functions which allow the programmer to directly perform bit manipulations upon FORTRAN variables. These functions are described in an Appendix entitled 'Facilities Used by the Compiler' in the FORTRAN IV (H) Compiler Program Logic Manual. The functions exist for the purpose of compiling the compiler (parts of which are written in FORTRAN).

IBM does NOT support these functions, therefore they must be used with some caution. An additional problem is that they must be specially SYSGENED into the compiler when it is created. Thus, a program which compiles under a compiler having this option may not compile when moved to another machine or under a new release of OS. On the other hand, once an object module has been created by such a compiler, it will always be good on any IBM 360 under OS. It is for this reason that the compiler built-in functions described below are included in this paper. A program which uses the functions has been successfully compiled and executed on the T&DS 360/95 (rel. 18) and 360/75 (rel. 19).

LAND

General form: ... = ... LAND (a,b)
where a and b are 4-byte integers

The value of LAND is obtained by anding the individual bits of the two arguments.

Example 1

```
I = 10
J = 3
K = LAND(I, J)
```

The value of K will be 2. I and J remain unchanged

LOR

General form: ... = ... LOR (a,b)
where a and b are 4-byte integers

The value of LOR is obtained by oring the individual bits of the two arguments.

Example 1

```
I = 10
J = 5
K = LOR (I, J)
```

The value of K will be 15. I and J remain unchanged.

LXOR

General form: ... = ... LXOR (a,b)
where a and b are 4-byte integers

The value of LXOR is obtained by exclusive oring the individual bits of the two arguments.

Example 1

```
I = 10
J = 11
K = LXOR (I, J)
```

The value of K will be 1. I and J remain unchanged.

LCOMPL

General form: ... = ... LCOMPL (a)
where a is a 4-byte integer

The value of LCOMPL is obtained by complementing all of the bits of the argument. That is, all of the 0 bits are made 1's and the 1 bits are made 0's.

Example 1

```
I = 15
J = LCOMPL(I)
```

Here, the value of I was set at hex '0000000F'. The value of J becomes hex 'FFFFFFF0'.

SHIFTL and SHIFTR

General form: ... = SHIFTL (J, K)

... = SHIFTR (J, K)

where J is a 4-byte variable and K is the number of bits to be shifted

The value of SHIFTL or SHIFTR is obtained by shifting the variable J to the left or right by the number of bytes specified by K.

Example 1

```
I = 15
J = SHIFTR (I, 2)
K = SHIFTL (I, 2)
```

The value of J will be 3, the value of K is now 60. I remains unchanged.

TBIT

General form: ... TBIT (A, K) ...

where A is a variable 4-bytes or less in length, and K specifies the position of the bit to be tested, where the leftmost bit is considered to be 0.

The value of TBIT is considered to be either .TRUE. or .FALSE., depending on whether the bit to be tested is on (1) or off (0).

Example 1

```
I = 1
IF(TBIT(I, 31)) GO TO 10
```

Execution of the above two statements would cause a branch to statement number 10, since bit 31 of the variable I was set to '1' prior to the test.

BITFLP

General form: V = BITFLP(V, K)

where V is a variable 8 bytes or less in length, and K indicates a bit position, where the leftmost bit is considered to be 0.

The value of BITFLP is obtained by inverting the Kth bit of the variable V.

Note: Only the variable used as argument to BITFLP will be changed. Assignment statements to another variable will be ignored.

Example 1

```
I = 14
I = BITFLP (I, 31)
```

The resultant value of I will be 15.

Example 2

```
I = 14
K = BITFLP (I, 31)
```

Here, the value of K remains unchanged, however the value of I will still become 15.

BITON and BITOFF

These two functions were supplied in order to give the capability to turn individual bits in a variable either 'on' or 'off'. Unfortunately, neither of these two functions will correctly compile. This is no particular loss, as one or more individual bits may be set either 'on' or 'off' via oring or anding against the appropriate mask.

Example 1

```
I = 3
K = LOR(K, I)
```

In the above, the rightmost two bits of the variable K will be set 'on'.

Example 2

```
DATA I/ZFFFFFFFC/
  .
  .
  .
K = LAND(K, I)
```

In the above, the rightmost two bits of the variable K will be set 'off'.

INVOKING the BUILT-IN FUNCTIONS

In order to use the functions described above, a special option must be passed to the FORTRAN H compiler via the PARM parameter of the EXEC card which invokes the compiler. It is coded as:

PARM.procstep = (..., XL, ...)

Example

```
//  EXEC FORTRANH,PARM = 'OPT = 2,LIST,XL'
```

BYTE MANIPULATION (FORTRAN G AND H)

Byte manipulation can be easily performed in both FORTRAN G and H*. No documentation of this by IBM is to be found in the FORTRAN IV Language Manual, however, documentation does exist in the form of a footnote in the FORTRAN IV [G and H] Programmers Guide. The vehicle used to accomplish byte manipulation is the LOGICAL * 1 variable. These variables can not only be used to indicate a .TRUE. or .FALSE. condition, but can also be used to store characters of information.

The documentation for this is hidden away in the section entitled "Extended Error Handling Facility" of the FORTRAN IV [G and H] Programmers Guide. Here, it is indicated that if a user-written error exit routine exists for certain I/O errors involving the attempted conversion of an illegal character, then the offending character will be passed to the exit routine in a LOGICAL * 1 variable. From this evidence, one can determine by experimentation that a great deal of flexibility is available in handling the LOGICAL * 1 variable.

Since the extended error handling facility is supported by IBM, and the use of the LOGICAL * 1 variable in the handling of bytes of data is an integral part of the facility, it must be presumed that this capability is a permanent feature of FORTRAN. In order to demonstrate the full range of facilities available in using the LOGICAL * 1 variable a FORTRAN program will be presented below. The statements will be in a logical sequence, interspersed with comments which will explain each capability. Each program statement will be dependent upon preceding statements. That is, use of a dimensioned variable will require a previous dimension statement, etc.

LOGICAL * 1 variables may be arranged in arrays

```
LOGICAL * 1 A(80), B(80), C(26), D(4,4)
```

They can appear in COMMON, EQUIVALENCE, and DATA statements.

```
REAL * 8 DATA(10)
COMMON D
EQUIVALENCE (DATA(1), A(1))
DATA C/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/
```

*FORTRAN H has considerably more flexibility than G in this matter.

They may be used in assignment statements with other variables of the same type. That is, the loop

```
      DO 10 I = 1,4
10    D(1, I) = C(I)
```

will copy the characters 'ABCD' into the first row of array D from array C. The following two statements are not allowable:

```
      I = 4
      D(3,3) = I
```

Two LOGICAL * 1 variables may be tested for equality or inequality (in FORTRAN H only).

```
      IF(C(1) .EQ. C(26)) GO TO 99
```

Execution of the above statement would not cause a branch to statement number 99, since 'Z' is not equal to 'A'.

In FORTRAN H (but not G) it is possible to successfully test for greater-than or less-than conditions between individual characters.

```
      IF(C(26).GT.C(1)) WRITE(6,1000)
1000  FORMAT (1X, 'Z IS GREATER THAN A')
```

They may be referenced indirectly in I/O statements

```
      READ(5,2000) DATA
2000  FORMAT (10A8)
```

or in a more direct manner such as

```
      WRITE(6,3000) A
3000  FORMAT (1X,80A1)
```

From this base, it is possible to easily perform any number of manipulations upon character-type data.