where the symbol $\cap$ denotes an AND operation and the symbol $*$ denotes a multiplication operation.

(2) The result of an OR operation with any number of Boolean variables is the same as the (arithmetic) addition of the x, y, z integer variables after the following test is made:

(a) If the sum is equal to zero, the result is correct;

(b) If the sum is larger than zero, the answer is a 1; i.e.

$$\begin{aligned} x + y + z &= A \cup B \cup C \qquad &\text{if } x + y + z = 0 \\ x + y + z &= 1 \qquad &\text{if } (x + y + z) \geqq 1 \end{aligned} \tag{2}$$

where the symbol $\cup$ denotes an OR operation and the symbol $+$ denotes an addition operation.

(3) The result of a NOT operation with a Boolean variable is the same as subtracting an integer variable x from 1; i.e.

$$\bar{A} = (1 - x) \tag{3}$$

because if $A = x = 1$, then $\bar{A} = 1 - 1 = 0$; and if $A = x = 0$, then $\bar{A} = 1 - 0 = 1$.

The FORTRAN program in Figure 1 illustrates the method presented. It simulates the logic of a full-adder as described by the following two Boolean functions:

$$L = K_1K_2 + K_1K_3 + K_2K_3 \tag{4}$$

$$M = \bar{L}(K_1 + K_2 + K_3) + K_1K_2K_3 \tag{5}$$

where K_1, K_2 and K_3 are the two input bits and previous carry to be added, L is the output carry, and M is the output sum. Integer variables were chosen for compatibility with the FORTRAN language.

```
C     EXAMPLE OF BOOLEAN SIMULATION
C     SIMULATION OF A FULL-ADDER
      DIMENSION  K(3)
C     INITIALIZE THE INPUT TRUTH TABLE TO ZERO
      DO 10 I=1,3
   10 K(I)=0
C     DERIVE THE TRUTH TABLE FOR THE SUM M, AND THE CARRY L.
      DO 110 I=1,8
      L = K(1)*K(2)+K(1)*K(3)+K(2)*K(3)
      IF(L) 20,30,20
   20 L = 1
   30 LT=K(1)+K(2)+K(3)
      IF(LT) 40,50,40
   40 LT=1
   50 M = (1-L)*LT + K(1)*K(2)*K(3)
      IF(M) 60,70,60
   60 M = 1
   70 PRINT 75,K(3),K(2),K(1),M,L
C     GENERATE THE NEXT INPUT COMBINATION
      K(3)=K(1)*K(2)*(1-K(3)) + (1-K(1)*K(2))*K(3)
      IF(K(3)) 80,90,80
   80 K(3)=1
   90 K(2) =K(1)*(1-K(2)) + (1-K(1))*K(2)
      IF(K(2)) 100,110,100
  100 K(2)=1
  110 K(1)=(1-K(1))
   75 FORMAT(10X,I3,I3,I3,4X,I3,I3)
      PAUSE
      END
```

FIG. 1

The AND and NOT operations are transformed to multiplication and subtraction operations as described in (1) and (3). The OR operation needs a control IF statement after the arithmetic addition is performed in order to restore the value of the variable to unity. This may be simplified by using a Function subprogram to calculate the result of the OR operation, thus eliminating the need for repetition of the IF statements. It was not done in this example because of the limitation of the FORTRAN compiler in the 1620 Model 1 computer where this program was checked out, and where the use of subprograms is not permitted.

M. MORRIS MANO
California State College at Los Angeles
Los Angeles, California

RECEIVED FEBRUARY, 1964

FURTHER REMARKS ON REDUCING TRUNCATION ERRORS

Recently Jack M. Wolfe [1] proposed the use of cascaded accumulators to evaluate a sum of the form $S = \sum_{i=1}^{N} y_i$ when N is large and all the y's are of roughly the same order of magnitude. His intention was to alleviate the accumulation of rounding or truncation errors which otherwise occurs when S is evaluated in the straightforward way illustrated by the following FORTRAN program.

```
 1  S = 0·0
 2  DO 4 I = 1, N
 3  YI = ···
 4  S = S + YI
 5  ····
```

The rounding or truncation in statement 4 could contribute to a loss of almost $\log_{10} N$ significant decimals in S. This would be important in those cases where the values of YI computed in statement 3 were correct to nearly full machine precision; otherwise the uncertainty in the YI's would swamp any additional error introduced in statement 4.

Of course, the simplest and fastest way to prevent such figure-loss is to accumulate S to double-precision. For example, in a FORTRAN IV program it would suffice to precede statement 1 above by the TYPE statement DOUBLE PRECISION S. The convenient accessibility of double-precision in many FORTRAN and some ALGOL compilers indicates that double-precision will soon be universally acceptable as a substitute for ingenuity in the solution of numerical problems.

In the meantime, programmers without easy access to double-precision arithmetic may be able to simulate it in the program above by a method far simpler than Wolfe's, provided they are using one of the electronic computers which normalize floating-point sums before rounding or truncating them. Among such machines are, for example, the I.B.M. 704, 709, 7090, 7094, 7040, 7044 and 360 (short word arithmetic).

The trick to be described below does not work on machines such as the I.B.M. 650, 1620, Univac 1107 and the Control Data 3600 which round or truncate floating-point sums to single precision before normalizing them.

In the following program S2 is an estimate of the error caused when S = T was last rounded or truncated, and is used in statement 13 to compensate for that error. The parentheses in statement 23 must not be omitted; they cause the difference (S−T) to be evaluated first and hence, in most cases, without error because the difference is normalized before it is rounded or truncated.

```
 1  S = 0.0
    S2 = 0.0
 2  DO 4 I = 1, N
 3  YI = ···
13  S2 = S2 + YI
    T = S + S2
23  S2 = (S−T) + S2
 4  S = T
 5  ····
```

Until double-precision arithmetic was made a standard feature of the FORTRAN language, the author and his students used this trick on a 7090 in FORTRAN II programs to perform quadrature, solve differential equations and sum infinite series.

REFERENCE:

1. WOLFE, J. M. Reducing truncation errors by programming. *Comm. ACM 7* (June 1964), 355–355.

W. KAHAN
University of Toronto
Toronto, Ontario, Canada

RECEIVED JULY, 1964

(*Pracniques are continued on page 48*)