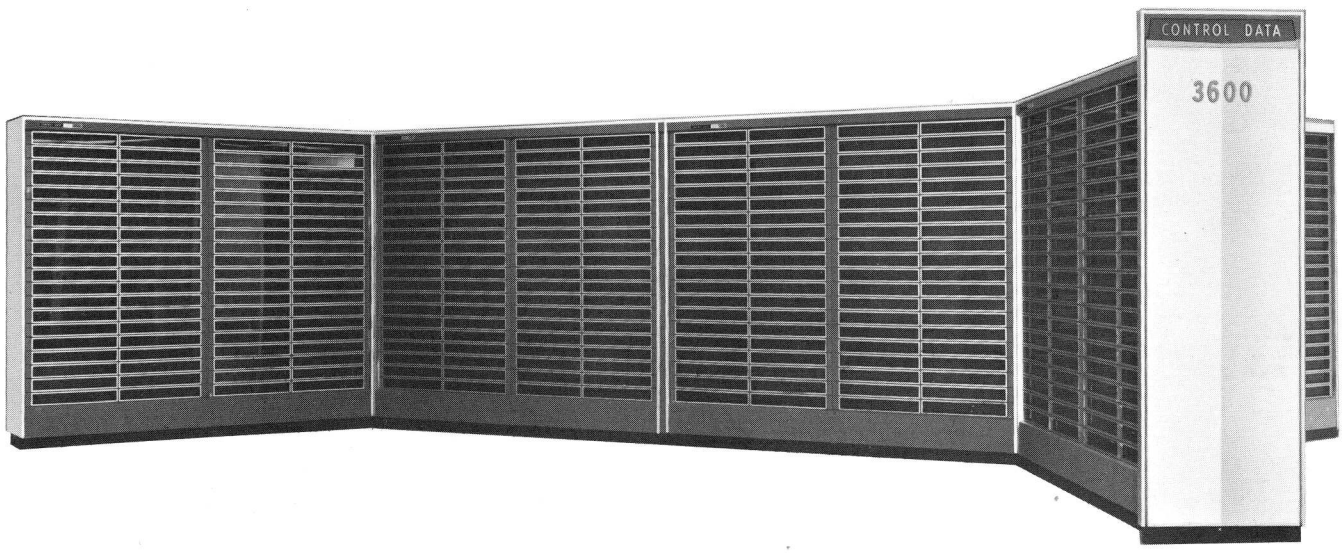
CONTROL DATA
3600 COMPUTER

3600

FORTRAN 63/GENERAL INFORMATION

CONTROL DATA 3600 COMPUTER



FORTRAN 63/GENERAL INFORMATION

CONTROL DATA CORPORATION
8100 34th Avenue South
Minneapolis 20, Minnesota

AUGUST 1962
Pub. No. 514

CONTENTS

FORTRAN-63 LANGUAGE

CONTENTS (Continued)

FORTRAN-63 IMPLEMENTATION NOTES

INTRODUCTION

The FORTRAN-63 language contains all of the features of its predecessor, FORTRAN-62, and forms an overset of the FORTRAN II language. This manual presents the complete FORTRAN-63 language with the new and more powerful features of the language presented in some detail.

The FORTRAN-63 compiler adapts current compiler techniques to the particular capabilities of the Control Data Corporation 1604 and 3600 computer systems. Emphasis has been placed on producing highly efficient object programs while maintaining the efficiency of compilation of FORTRAN-62. This manual describes some of the more important implementation features -- especially those pertaining to index functions and the interpretation of arithmetic statements.

The FORTRAN-63 compiler is part of a complete operating system -- the CO-OP Monitor System for the 1604, and the Master Control System for the 3600.

For the reader who is unfamiliar with the FORTRAN language or its implementation on Control Data Corporation equipment, the following publications may be found helpful: FORTRAN Autotester, Publication No. 186A; FORTRAN-62 Reference Manual, Publication No. 506; CO-OP Monitor Programmer's Guide, Publication No. 508.

CHARACTER SET

A character is defined as that mark obtained by striking one key on a typewriter or keypunch machine.* The FORTRAN-63 character set comprises:

26 letters (A through Z)

10 digits (0 through 9)

space (blank)

10 special characters + − * /) (= , . $

The special character = denotes replacement. $ is used as a statement separator. Thus statements without statement numbers that are conventionally written one per line may be compacted by using the $ operator.

In addition to the conventional arithmetic symbols +, −, *, /, the following compound symbols are provided for use in arithmetic/conditional statements.

MATHEMATICAL NOTATION	MEANING	FORTRAN-63 SYMBOL
$[\ ]^n$	exponentiation	**
=	equal	.EQ.
$\neq$	not equal	.NE.
>	greater than	.GT.
$\geq$	greater than or equal to	.GE.
<	less than	.LT.
$\leq$	less than or equal to	.LE.
$\wedge$	logical and	.AND.
$\vee$	logical or	.OR.
$\neg$	not	.NOT.

* "key" includes the space bar.

CONSTANTS

The mode of a constant is determined from context. Let

n	be a concatenated set of the digits 0-9
s	be a scalar
o	be a concatenated set of the digits 0-7
h	be a length of a Hollerith field
f	be a Hollerith field
R	be a single precision floating point number

Then:

n	denotes either an integer or a statement number
n.n n. .n nEs n.nEs .nEs n.Es	denote single precision floating point numbers
n.nD n.D .nD nDs n.nDs .nDs n.Ds	denote double precision floating point numbers
oB	denotes an octal integer
hHf	denotes Hollerith literals, left justified with blank fill
(R,R)	denotes a complex constant

E and D are floating point designators, B is the octal designator, and H is the Hollerith designator.

STATEMENTS

Replacement Statement

The replacement or arithmetic statement specifies a calculation resulting in a value. In the statement

a = b

a is any variable name, simple or subscripted, and b is any arithemtic expression. The right-hand side of the statement must obey the mode rules of arithmetic expressions.* The left-hand side of the statement may be of a mode different from that of the right-hand side. Briefly, the expression b is evaluated and its value is assigned to the variable name a. The significance of the operator = is replace by.

Declarative Statements

The declarative statements of FORTRAN-63 are described below. All declarative statements (except FORMAT) must precede the first executable statement of a program.

PROGRAM name		
SUBROUTINE name	(fp_1 , fp_2 ,..)	
FUNCTION name	(fp_1 , fp_2 ,..)	where fp_i is a formal parameter

These three statements form the left bound of programs or subprograms (the END statement forms the right bound).

An identifier followed by a parenthesis is assumed to be a subroutine if the identifier is not declared in a DIMENSION statement.

The naming conventions for functions and subroutines are the same as those for variables. That is, the mode may be declared in a TYPE statement (page 6). Or, in the absence of a TYPE REAL or TYPE INTEGER statement, the mode is determined by the first letter of the identifier (A through H and O through Z for REAL; I through N for INTEGER).

The mode of the arguments of a function or subroutine is determined by the declared TYPE of the arguments. If the argument is an expression, the mode is determined by the rules of precedence governing arithmetic with mixed operands.*

---

*Arithmetic Statements, page 20.

In the 3600 system, any single (sub) program or subroutine will be executed from a single bank. The parameters or data associated with the (sub) program or subroutine may occupy any bank. (See COMMON below.)

DIMENSION
COMMON
EQUIVALENCE

From these statements (together with TYPE) are derived the data storage requirements. In the DIMENSION statement, provision is made for declaring array dimensions as formal parameters. Thus the statement DIMENSION A(m,n), is permissible if A,m and n are formal parameters.

Bank allocation for data in the 3600 must be declared in the COMMON statement. The format for designating bank is:

$$\text{COMMON}/I_1\ (B)/\text{LIST}/I_2\ (B)/..$$

where I is a block identifier and B is a bank designator ($0 \leq B \leq 7$). If B is not stated, it is assumed to be bank zero. The bank designator is ignored in the 1604 system.

The block identifiers distinguish between two types of COMMON block storage: Labeled and Numbered*. Data may be prestored in labeled COMMON (via the DATA statement), but not in numbered COMMON. The block identifiers may be up to eight characters in length. For labeled COMMON, the identifier characters are alphanumeric, with the first character alphabetic. For numbered COMMON, the identifier characters are all numeric.

* Sometimes referred to as Blank COMMON.

The COMMON statement may be written in a variety of forms. For example:

	COMMON A,B	identical in meaning
numbered	COMMON //A,B	
labeled	COMMON /LIDO3/A,B	
numbered	COMMON /10/A,B	/10/ interpreted as /10(0)/ in the 3600
numbered	COMMON /10(2)/A,B	/10(2)/interpreted as /10/ in the 1604

TYPE Declarations

FORTRAN-63 is designed to accommodate eight distinct modes of arithmetic and/or operands. The general form of the type declaration is:

TYPE name (λ) (identifier list)

where λ is the element length of entities appearing in the list. λ is either w words or f bits per element and is of the form w or /f.

Five of the types are standard; the routines or instructions required for these arithmetic modes are provided with the system. λ does not appear explicitly in the standard types. The remaining three types are arbitrary,* both in mode and execution.

Formats for the standard type declarations are:

0. TYPE INTEGER (identifier list). Integer variables, array names, and integer functions.
1. TYPE REAL (identifier list). Single precision floating point variables, array names, and functions whose value is real.

---

* Arbitrary Modes of Arithmetic, page 24.

2. TYPE DOUBLE (identifier list). Double precision floating point variables, array names, and functions.
3. TYPE COMPLEX (identifier list). Complex variables (represented by two reals), array names, and functions.
4. TYPE LOGICAL (identifier list). Logical variables, array names, and functions.

Identifiers not declared in TYPE statements are REAL or INTEGER according to the first letter of the identifiers (for REAL: A through H and O through Z; for INTEGER: I through N).

For types 0 and 1, the element length is one word; for types 2 and 3, the element length is two words; for type 4, the element length is one word for non-dimensioned variables, or one bit for dimensioned variables.

Thus a non-dimensioned variable, Y, declared TYPE LOGICAL, will have one word allocated to it. A dimensioned variable, Z(96), declared TYPE LOGICAL, will have two words allocated to it, each bit being an element of the array Z.

FORMAT Statement

This statement controls formating to and from external media. The conventional FORMAT conversions are:

Em.n	denoting E type single precision
Fm.n	denoting F type single precision
Im	denoting Integer conversion
Om	denoting Octal integer conversion
Am	denoting Alphanumeric conversion
mX	denoting m spaces (no conversion)
mH	denoting m BCD characters (no conversion)

Four conversion types have been added:

DEm.n	denoting E type double precision
DFm.n	denoting F type double precision

C(αm.n, αm.n)	denoting complex conversion where α may be either E or F conversion
Sm.n	denoting a mode of conversion determined by the TYPE of variable which corresponds to this designator. m denotes length of field. n is a free parameter to be utilized by the conversion routine provided for variables of this TYPE.*

EXTERNAL Statement

This statement declares the identifiers following to be function names. It is required only if those names are used exclusively as actual parameters to subroutine calls. This statement replaces the function of an F card.

DATA Statement

The DATA statement assigns constant values to variables at load time. Its format in FORTRAN-63 is:

DATA (identifier = value list), (identifier = value list),.. where identifier is a single

1) non-subscripted variable, (I = 7)
2) array variable with constant subscripts, (A(1)=7.5)
3) array name, (A=1.,2.,3.)
4) array element with constant quantifiers,

((A(I), I=1,3) = 1.0,2.0,3.0)

The value list is either a single constant (for 1 and 2) or a set of constants whose number is equal to the number of elements in the named array (for 3 and 4). The conversion is determined by the mode of the value list and includes the six types enumerated under Constants on page 2.

---

* Arbitrary Modes of Arithmetic, page 24.

CONTROL Statements

These statements direct the flow of the program.

IF (a) n_1 , n_2 , n_3

For negative values of the arithmetic expression a, a jump to n_1 is executed; for zero value, a jump to n_2 ; and for positive values, a jump to n_3. n_1 , n_2 , n_3 are statement numbers.

IF (ℓ) n_1 , n_2

This is the logical IF statement. A jump to n_1 is executed if the logical expression ℓ is TRUE, and to n_2 if it is false. If ℓ is an arithmetic expression, non-zero is assigned TRUE and zero FALSE.

EXAMPLE: If a*b is greater than c, and if either d is greater than zero or e equals f, go to n_1 . If not, go to n_2 .

In FORTRAN-63 this is written:

IF (((A*B).GT.C).AND.D.OR.(E.EQ.F)) n_1 , n_2 .

IF DIVIDE $\left\{\begin{matrix}\text{FAULT}\\ \text{CHECK}\end{matrix}\right\}$ n_1 , n_2

Either statement checks the status of the divide fault indicator and executes a jump to n_1 if it has been set (and turns it off), or a jump to n_2 if it has not been set.

IF OVERFLOW FAULT n_1 , n_2

This statement checks the status of the overflow fault indicator and executes a jump to n_1 if it has been set (and turns it off), or a jump to n_2 if it has not been set.

IF EXPONENT FAULT n_1 , n_2

This statement executes a jump to n_1 if exponent overflow has occurred, or a jump to n_2 if it has not occurred.

SENSE LIGHT i
IF (SENSE LIGHT i) n_1 , n_2
IF (SENSE SWITCH j) n_1 , n_2

For the 3600, i may assume the values 1 through 48, corresponding to the number of bits in the D (display) register. j may be 1 through 6, corresponding to six console switches. In the 1604, the functions are simulated with programmed binary flip-flops.

In either case SENSE LIGHT i (for $i \neq 0$) sets the light i flip-flop, and SENSE LIGHT 0 turns off all sense lights. If light i is set (ON) when it is tested, it is turned OFF and a jump to n_1 is executed. Similarly, a jump to n_1 is executed if SWITCH j is set. (In the 1604, the setting of FORTRAN switches is a Monitor function.)

DO n i = m_1 , m_2 , m_3

The various properties below describe the implementation of the DO statement in FORTRAN-63.

1) m_1 , m_2 , m_3 are either fixed point constants or integer non-subscripted variables.
2) m_1 , m_2 , m_3 assume positive values only.
3) The values of m_1 , m_2 , m_3 are assumed to remain constant until the DO is satisfied.
4) If $m_1 > m_2$ (initially) the loop is not executed.
5) Within the range of statements bounded by the DO statement, i cannot appear in any statement assigning a new value to it.

6) i has an identity and value local to the statements in the range of the DO statement if it is

   a) Not used as an operand.

   b) No transfers out of the (physical)range of the DO exist.

   Otherwise, its definition and value are global.

7) Nesting of DO statements is permitted to reasonably arbitrary levels.

GO TO n
GO TO n, (n_1 , n_2 , .., n_m)
GO TO (n_1 , n_2 , .., n_m),i
ASSIGN i TO n
CONTINUE
PAUSE n
STOP n

These statements are implemented according to their conventional definitions. PAUSE n and STOP n will be implemented according to local requirements (return to monitor control, etc.).

Input/Output Statements

The following statements are implemented according to their conventional definitions.

FORMAT

READ n,L

READ INPUT TAPE i,n,L

READ TAPE i,L

PUNCH n,L

PRINT n,L

WRITE OUTPUT TAPE i,n,L

WRITE TAPE i,L

END FILE i

REWIND i

BACKSPACE i

i is a logical unit number
n is a FORMAT statement number, a simple variable, an array identifier or a formal parameter
L is an output variable list

Four alternative forms for the READ/WRITE statements given above are:

READ (i,n)L	Alternate form of READ INPUT TAPE i,n,L and READ n,L
READ (i)L	Alternate form of READ TAPE i,L
WRITE (i,n)L	Alternate form of WRITE OUTPUT TAPE i,n,L, PUNCH n,L and PRINT n,L
WRITE (i)L	Alternate form of WRITE TAPE i,L

The new data transfer statements which have been added are described below.

BUFFER IN (i,p) (A,B)
BUFFER OUT (i,p) (A,B)

i is a logical unit number ($1 \leq i \leq 49$)

p is the parity key

A and B are simple or subscripted variable names defined in the same block of COMMON or in unique storage. The address of B must be greater than that of A.

BUFFER IN reads (with buffering) one physical record from logical unit i into memory locations corresponding to A through B. The record is read in odd parity if p = 0, or in even parity if p = 1.

BUFFER OUT writes (with buffering) the words from the memory locations corresponding to A through B into one physical record on logical unit i. The record is written in odd parity if p = 0, or in even parity if p = 1.

IF (UNIT, i) n_1 , n_2 , n_3, n_4

i is a logical unit number

n_j are statement numbers

This statement checks the status of previously initiated buffered operations. It also checks for certain errors. The statement jumps control to:

n_1 if the previously initiated buffered operation is not complete.

n_2 if that operation is complete and NO errors occurred.

n_3 if that operation is complete and an EOF or EOT was encountered.

n_4 if that operation is complete and parity or buffer length errors occurred.

IF (EOF, i) n_1 , n_2
IF (IOCHECK, i) n_1 , n_2

i is a logical unit number

n_1 , n_2 are statement numbers

IF (EOF) checks the previous READ or WRITE operation performed on unit i for an end-of-file (end-of-tape) during that operation. If an end-of-file or end-of-tape was encountered, statement n_1 is executed. If not, statement n_2 is executed.

IF (IOCHECK) checks the previous READ or WRITE operation performed on unit i for parity errors. If a parity error occurred, statement n_1 is executed. If not, statement n_2 is executed.

NOTE: The IF (EOF) and IF (IOCHECK) statements may be used in any order, but during an even parity READ check, an end-of-file will produce both EOF and IOCHECK error indications.

DECODE (i,n,v)L
ENCODE (i,n,v)L

- i is the unit record length in characters
- n is a FORMAT statement number
- v is a simple variable or an array name
- L is a variable list

Two statements, ENCODE and DECODE, have been added to the language to permit the transfer of information from an array to list variables and vice versa.

DECODE: the information in array v is converted according to FORMAT statement n and stored in the list variables.

ENCODE: information in the list variables is converted according to FORMAT statement n and stored in array v.

FORTRAN-63

IMPLEMENTATION NOTES

INDEXING

One of the principal considerations in the implementation of FORTRAN-63 is the efficiency of the object code, particularly in regard to indexing. Although subscript expressions of arbitrary form and type are permitted, emphasis is placed on the efficient implementation of the standard subscript forms. These are:

C

C*i

i ± d

i

C*i ± d

The method employed by the compiler involves the concept of the index function.

Consider the three-dimensional array A(I,J,K). The address of an element of A, A(i, j, k), is given by

Locn A + (i − 1) + (j − 1)*I + (k − 1)*I*J,

where i, j, k are subscript variables, and I and J are the values of the first and second dimensions provided by a DIMENSION statement.

The above equation can be written:

Locn A − (1 + 1 * I + 1*I*J) + (i + j*I + k*I*J)

Let

Locn A	be the base address
−(1 + I + I*J)	be the constant addend
(i + j*I + k*I*J)	be the index function, ϕ_I

The address of A (i,j,k) for a particular set of values of i, j, k is given by an address (established by adding the constant addend to the base address) modified by a B-box containing the value of the index function. Furthermore, if and when i, j, k change, only the value of the index function need be changed, irrespective of the number of appearances of A (i,j,k) in the

program. This use of index registers for address modification is the principal way in which indexing efficiency is obtained.

An index function is generated for every unique combination of subscript variables/multipliers. For a change in i, the form of its evaluation is:

$$(i_{new} - i_{old}) \text{ (multiplier)} + \phi_I \rightarrow \phi_I$$

thus necessitating at most one multiply per index function per change. The multiply is replaced by an add if either the first or the second factor in the product is one. The first factor is tested at execute time; the second is determined at compile time.

Subscript variables occurring as quantifiers in DO statements have certain restrictive properties that can be utilized by the indexing algorithms. Furthermore, the nature of the legitimate DO statement is such that, in many cases, all information pertinent to optimum indexing is known at compile time. Thus, in a DO loop, or a nest of DO loops, if no branches out of the range of the DO exist, then all indexing, counting, and testing is done in the index registers. The following example illustrates the application of these criteria.

1	5		7	72
			DIMENSION A(40,40),B(40,40),C(40,40) DO 12 I = 1,40 DO 12 J = 1,40 C(I,J) = 0.0 DO 12 K = 1,40	
	12		C(I,J) = C(I,J) + A(I,K)*B(K,J)	

The above source program sequence is evaluated by the compiler as shown on the following page.

	ENI	6	39		Imax − 1
	ENI	1	1641		Imin + (Jmax + 1)*40
	ENI	2	1641		Imin + (Kmax + 1)*40
A1	ENI	5	39		Jmax −1
	ENI	3	81		(Kmax + 1) + Jmin *40
	INI	1	−1600		(Jmin − Jmax −1)*40
A2	ENA	0	0		
	STA	1	C-41		
	ENI	4	39		Kmax − 1
	INI	2	−1600		(Kmin − Kmax − 1)*40
	INI	3	−40		(Kmin − Kmax − 1)
A3	LDA	2	A-41		
	FMU	3	B-41		
	FAD	1	C-41		
	STA	1	C-41		
	INI	2	40		
	INI	3	1		
	IJP	4	A3		
	INI	1	40		
	INI	3	40		
	IJP	5	A2		
	INI	1	1		
	INI	2	1		
	IJP	6	A1		

ARITHMETIC STATEMENTS

The translation of arithmetic expressions by FORTRAN-63 follows the classical rules of precedence: exponentiation, multiplication-division, addition-subtraction. The ordering is left to right between parenthesized expressions. Within an expression, the compiler will generate the most efficient ordering of operations governed by the rules of precedence and commutativity.

Twelve instruction types are generated by the translator (exclusive of the index commands):

0	Load	operand
1	Load negative	operand
2	Add	operand
3	Subtract	operand
4	Multiply	operand
5	Divide	operand
6	Complement	accumulator
7	Power	---
8	Call	function
9	Parameter	---
10	Convert	accumulator or operand
11	Store	operand

Instructions are generated independently of the arithmetic mode and the type of operand. The appropriate machine order, or a jump to a routine which executes the particular intent, then replaces the generated instruction type.

The arithmetic mode of an expression is determined as follows: The arithmetic mode corresponds to the highest order type of any operand within the expression. The order of the types, from lowest to highest, is:

TYPE Integer

TYPE Real

TYPE Double

TYPE Complex

As an example, given TYPE declarations

a, b TYPE Integer

c TYPE Real

d TYPE Double

e TYPE Complex

The expression ((((a + b) + c) + d) + e) is evaluated as follows:

LOAD a	
ADD b	(integer add)
CONVERT	(a + b) in accumulator to Real
ADD c	(floating add)
CONVERT	((a + b) + c) in accumulator to Double
ADD d	(double floating add)
CONVERT	(((a + b) + c) + d) in accumulator to Complex, i.e., (Real, 0)
ADD e	(complex add)

The word CONVERT indicates that the conversions are effected by jumping to appropriate subroutines. In the 1604, ADD d is performed by a double precision floating add subroutine. In both the 1604 and the 3600, ADD e is performed by a complex add subroutine.

LOGICAL ARITHMETIC

To provide consistency with earlier FORTRAN systems, a 48-bit Boolean arithmetic has been implemented that is not dependent on the column 1 B-designator. In this arithmetic, the operands must be real or integer, and the operators .AND., .OR., .NOT. replace * + − , respectively. A statement previously written as:

1	5		7	72
B			A = B + C * (− D)	

is written in FORTRAN-63 as:

1	5		7	72
			A = B .OR. C .AND. .NOT. D	

A logical expression is a proposition involving relational operators and logical connectors. Relational operators are:

$$= \quad \neq \quad > \quad < \quad \geq \quad \leq$$

A relation consists of two arithmetic expressions connected by a relational operator, and its value is either true or false.

A logical connector is either $\wedge$ or $\vee$ representing AND (conjunction) and OR (disjunction), respectively. The unary operator $\neg$ (NOT) permits negation of propositions.

These properties of relational operators and logical connectors are used in the two-branch (logical) IF statement. This statement has the general form:

IF ((α rel α) L (α rel α) L . .) n_1, n_2

where α is an arithmetic expression

rel is a relational operator

L is a logical connector

n_1 is a statement number, executed if the proposition is TRUE

n_2 is a statement number, executed if the proposition is FALSE

To express the values TRUE and FALSE, the compiler implements the expression in relation to the numbers 1 (for true) and 0 (for false).

Let ρ be a relation such that if a is related to b as demanded by ρ, the a ρ b is true and a true, 1, branch is executed; if the converse obtains, then a ρ b is false and a false, 0, branch is executed. Further, let

a, b designate arithmetic expressions

and define

JNE jump if (expression) not equal to zero

JEQ jump if (expression) equal to zero

JLT jump if (expression) less than zero

JGE jump if (expression) greater than or equal to zero

The table shows the branching-logic for "Is a ρ b?"

Logical Relation	Expression Evaluation	True jump (1)	False jump (0)
a	a	JNE	JEQ
$a = b$	a - b	JEQ	JNE
$a \neq b$	a - b	JNE	JEQ
$a < b$	a - b	JLT	JGE
$a \geq b$	a - b	JGE	JLT
$a > b$	$-a + b$	JLT	JGE
$a \leq b$	$-a + b$	JGE	JLT

Now recall the example given for the logical IF statement on page 8. That is:

If $((a * b) > c) \wedge d \vee (e = f))$ is true, go to n_1; if not, go to n_2.

The compiler implementation, based on the branching logic described above, is:

	Expression Evaluation	True Jump	False Jump
	−a*b+c	JLT α_1	JGE α_2
α_1	d	JNE n_1	JEQ α_2
α_2	e-f	JEQ n_1	JNE n_2

In expressions of the form $\neg(\ell_1 \wedge \ell_2 \vee \ell_3)$ the negation is taken inside the parentheses: $((\neg \ell_1 \vee \neg \ell_2) \wedge \neg \ell_3)$. Here ℓ denotes branch; $\neg \ell$ denotes reverse branch.

The arithmetic in a logical statement may be of any mode; all testing, however, is done on single precision values (values appearing in the accumulator). Thus, to test the real and imaginary parts of a complex expression C for non-zero, an appropriate FORTRAN-63 logical expression would be

$$(C \wedge (C*(0., -1.))$$

ARBITRARY MODES OF ARITHMETIC

FORTRAN-63 provides the capability for generating calls to arithmetic library routines from criteria based on the basic arithmetic symbols. Further, all control, input-output, data allocations, and indexing is provided automatically from considerations derived from the declarative statements.

To introduce a new type of arithmetic, no changes are required in the compiler. The user indicates the type in a TYPE declaration and provides the library routine which executes the instruction types generated. No more than three types (other than the standard five) are permitted in any one program or subprogram. The same type, with different w and f designations, may be declared arbitrarily many times. The compiling technique is the following.

When a TYPE statement which is not standard is encountered, a call is generated to a library routine whose identifier is name. The space requirements are calculated from w or f (the length of an element) and the DIMENSION statements; w and f also provide the length-per-element parameter required by the indexing and input-output routines.

When an operand declared to be TYPE name (non-standard) occurs in an arithmetic statement, the mode of arithmetic for that statement becomes that type. Mixed arithmetic is permitted with standard operands. For each instruction type generated by the translator a jump to routine name is placed in the object code. Associated with each jump are the parameters w or f, operand, instruction type.

In addition to the 12 instruction types, two I/O jumps are generated and invoked whenever Sm.n is encountered in a FORMAT statement. The particular I/O jump is determined from the type of operand associated with the S designation, and the statement type (INPUT or OUTPUT).

Thus, each library routine for a non-standard TYPE consists of 12 (instruction types) + 2 (I/O) sections, each implementing a particular instruction type. If some of the operations are not defined, the number of sections can be correspondingly limited.

An example of an arbitrary mode of arithmetic is derived from J. H. Wegstein and W. W. Youten, "A String Language for Symbol Manipulation Based on ALGOL 60," Communications of the ACM, Vol. 5, No. 1, p. 58.

This subroutine translates a fully parenthesized arithmetic expression into Lukasiewicz' s parenthesis-free notation. The input string is in S(1) through S(n), the output in P(i + 1) through P(n).

Examples of output by the routine are:

IN	OUT
(((a + b)−c)*d)	* − + abcd
(a + (b − (c*d)))	+ a − b * cd
(((a + b) − (c*d))/e)	/ − + ab * cde
(((a + (b − c))*((d/e)+f))−g)	− * + a − bc + /defg
((a + b) * (c − d))	* + ab − cd

Source Program

```
1        5         7                                                    72
         SUBROUTINE POLISH (S,P,T,N)
         TYPE BYTE (/6) S,P,T
         DIMENSION S(N), P(N), T(N)
         COMMON,I
         K = 1 $ I = N $ J = N

       1 IF (S(J)=1H))8,2
       2 IF (S(J)=1H+∨S(J)=1H−∨S(J)=1H*∨S(J)=1H/)3,4
       3 T(K)=S(J) $ K=K+1 $ GO TO 10
       4 IF (S(J)=1H( )5,6
       5 P(I)=T(K−1) $ K=K−1 $ GO TO 7
       6 P(I)=S(J)
       7 I=I−1
       8 IF (J=1)9,10
       9 RETURN
      10 J=J−1 $ GO TO 1 $ END
```

The BYTE library routine for this routine would require four sections: Load, Convert, Subtract, Store; the codes would be:

Load:	Load the 6 (f) bits of the operand into the lower part of the accumulator.
Convert:	Store the 6 (f) lower bits of the operand in erasable storage.

Subtract:	Fixed point subtract the operand (erasable storage).
Store:	Extract the 6 (f) lower bits of the accumulator and store them in the operand location.

COMPILER STRUCTURE

The FORTRAN-63 compiler consists of a translator and an assembler. The translator reads the source language (once) from an input tape. The translator and the assembler operate on memory contained lists and communicate with each other via these lists. The output (object program) is relocatable binary card images on magnetic tape.

Compilation proceeds subprogram by subprogram, each subprogram being independently compiled. If, during the compilation process, the lists generated for a subprogram require more than the available core, the excess is recorded on and read back from a scratch tape. Only for very large subprograms will a scratch tape be required.

Other publications concerning programming and programming systems for the Control Data Corporation 3600, 1604, and 1604-A Computers are:

PERT	#133
1604 Programming Manual	#167A
Fortran Auto Tester	#186A
Satellite Programming	#187
Fortran-62 Reference Manual	#506
CO-OP Monitor/Programmer's Guide	#508
CO-OP Monitor/Operator's Guide	#509
CODAP-1 Reference Manual	#510
CDM2 Linear Programming System	#511
3600 Preliminary Reference Manual	#523

CONTROL DATA SALES OFFICES

ALBUQUERQUE, N. M., 937 San Mateo, N.E., Phone 265-7941

BEVERLY HILLS, CALIF., 8665 Wilshire Boulevard, Phone OL 2-6280

BIRMINGHAM 13, ALA., 16 Office Park Circle, Phone TR 1-0961

BOSTON, MASS., 594 Marrett Road, Lexington, Mass., Phone VO 2-0002

CHICAGO, ILL., 840 South Oak Park Avenue, Oak Park, Ill., Phone 386-1911

CLEVELAND, OHIO, Center Building, 46 West Aurora Road, Northfield, Ohio, Phone 467-8141

DALLAS 35, TEXAS, 2505 West Mockingbird Lane, Phone FL 7-7993

DAYTON 29, OHIO, 10 Southmoor Circle, Phone 298-7535

DENVER 3, COLORADO, 655 Broadway Building, Phone AC 2-8951

DETROIT, MICHIGAN, 12800 West Ten Mile Road, Huntington Woods, Michigan

HOUSTON 27, TEXAS, 4901 Richmond Avenue, Phone MA 3-5482

ITHACA, NEW YORK, Cornell University, Rand Hall, Phone AR 3-6483

KANSAS CITY 6, MISSOURI, 921 Walnut Street, Phone HA 1-7410

MINNEAPOLIS 20, MINN., 8100 34th Avenue South, Phone 888-5555

NEWARK, NEW JERSEY, Terminal Building, Newark Airport, Phone MI 3-6446

NORFOLK 2, VIRGINIA, P.O. Box 1226, Phone 341-2245

ORLANDO, FLORIDA, P.O. Box 816, Maitland, Florida, Phone 647-7747

SAN FRANCISCO, CALIF., 885 North San Antonio Road, Los Altos, Cal., Phone 941-0904

WASHINGTON 16, D.C., 4429 Wisconsin Avenue N.W., Phone EM 2-2604

WASHINGTON 10, D.C., 1515 Ogden Street N.W., Phone RA 6-4983

CONTROL DATA
CORPORATION

8100 34TH AVENUE SOUTH, MINNEAPOLIS 20, MINNESOTA

PUB. NO. 514