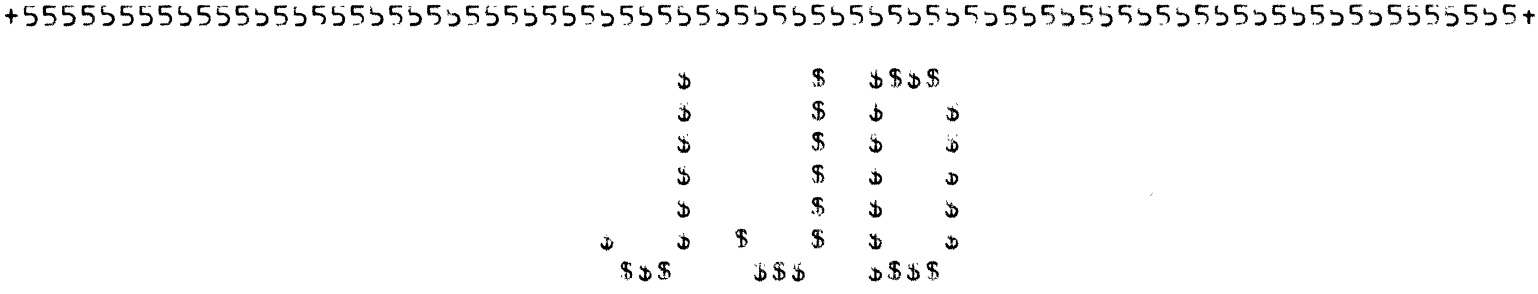
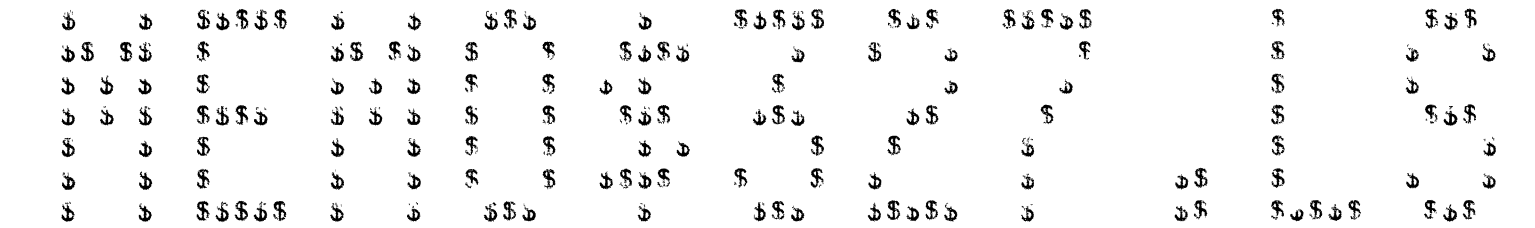


USER=JJD  QUEUE=LPT  DEVICE=@LPB1
SEQ=6  QPRI=127  LPP=63  CPL=80  COPIES=1  LIMIT=64

CREATED:  7-NOV-77  15:35:16
ENQUEUED:  6-DEC-77  14:21:14
PRINTING:  6-DEC-77  14:33:54

PATH=:PDU:MEMO:MEMO$327.LS



AOS XLPT REV 01.00

To: all interested

From: Walter Wallach

Subject: FORTRAN S-Language (Revised)

Date: 7/Nov/77 Memo No. 327

Abstract:

This memo revises Memo 311, the FORTRAN S-Language Specification. A Floating Point Data Error Exception is introduced to capture invalid or unnormalized floating point data.

# 1 FORTRAN S-Language

In the following description of the SPRINT FORTRAN dialect, type definitions, instructions definitions, and most instructions are written in SPL.

## 1.1 Operand Types for the FORTRAN S-Language

Instructions are of two basic types: arithmetic and control. The arithmetic instructions define operations on one, two or three operands ranging from simple assignment (move) to COMPLEX arithmetic and transcendental functions. The control instructions each define some decision and branch. The decision may be based on a comparison or an arithmetic result, or it may be unconditional (GOTO or CALL). The branch may be relative to the I-stream or absolute (relative to the start of some procedure object).

### 1.1.1 Relative branches

A relative branch instruction specifies as an operand a "relative offset" syllable. This syllable is a $k$-bit literal denoting a signed, integral, nibble granular offset relative to the current PC (i.e., the address of the instruction containing the relative branch. The $k$-bit relative offset syllable is multiplied by four to obtain a bit-granular offset and then sign-extended to 32 bits. This value is then added (2's complement addition) to the offset portion of the current PC. Interpretation resumes at this new I-stream location.

Note that a relative branch is always intra-Procedure Object, since only the offset of the PC is modified.

### 1.1.2 Offset Branch

An offset branch is an intra-Procedure Object branch specified relative to the start of the current Procedure Object. An offset name operand is evaluated (like any other name), yielding an integral offset (signed or unsigned, as specified in the associated Name Table Entry). This offset value represents a bit-granular offset which is added to the offset portion of the current Procedure Object Pointer (POPTR). The result of this addition replaces the PC.

### 1.1.3 Data Representations

The FORTRAN language defines two classes of storage which, by definition, may never overlap in a "standard conforming" program.

All addresses are bit granular, locating containerized data. All lengths are bit lengths. In general, operand lengths are specified in the length field of the operand's NTE. In some cases, however, the opcode may imply a length. Data representation is always implied in the opcode. No type checking is performed, the type field of the NTE is ignored.

#### 1.1.3.1 CHARACTER Data

The first storage class is that of a CHARACTER machine. This machine is implemented as an 8-bit byte containerized machine, where bytes reside at addresses divisible by 8. The standard insures that CHARACTER data may not be EQUIVALENCED to non-character data.

In most cases, the length of a character string is known at compile time. When the length may vary (character parameters), length may be specified "as a name" in the Name Table. In these cases, length should be passed as an INTEGER parameter.

#### 1.1.3.2 Classical FORTRAN Data

The second storage class is that of a "classical" FORTRAN machine, a containerized storage machine with the expected FORTRAN operations. We implement this class as a 16-bit word machine (addressed in a bit-granular fashion) and utilize data representations compatible with IBM 360 and ECLIPSE data types. There are no restrictions concerning word alignment other than the policy that all containers will begin on a bit address divisible by 16.

1.1.3.2.1 INTEGER

All integers are 2's complement and occupy one 32-bit storage container. All INTEGER operations are 2's complement operations. Overflows are detected on completion of an operation that overflows the container and prior to storing the result.

INTEGER data is addressed at its left-most (i.e. low-address) end, right-justified and sign-filled on the left, truncated on the left (see discussion of "Execution Exceptions" below).

```
TYPE integer IS BIT(32) TAKEN AS
        -(2**31)..( (2**31)-1 );
```

1.1.3.2.2 INTEGER*2

INTEGER*2 is a half-size 2's complement integer. It differs from INTEGER only in length.

INTEGER*2 data is addressed at the left-most (i.e., low address) end, right-justified and sign-filled on the left, truncated on the left (see discussion of exceptions).

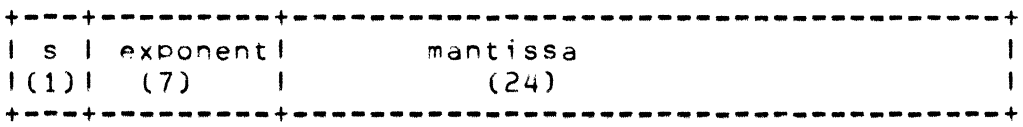
```
TYPE integer*2 IS BIT(16) TAKEN AS
        -(2**15)..( (2**15)-1 );
```

1.1.3.2.3 REAL

All single precision REAL values occupy a single 32-bit storage container and are "IBM format" binary floating point (i.e., an algebraic sign bit followed by a 7-bit, excess-64 hexadecimal exponent, followed by a 24-bit hexadecimal normalized mantissa).

REAL data is addressed at the left-most (low-address) end, left-justified and zero-filled on the right, truncated on the right.

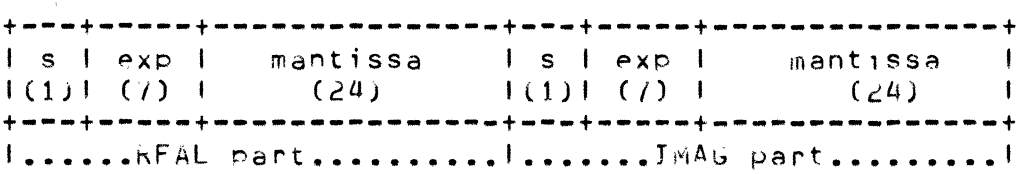
```
TYPE real IS BIT(32) TAKEN AS RECORD
        sign        :(0=positive, 1=negative);
        exponent    :BIT(7) TAKEN AS 0..127;
        mantissa    :BIT(24) TAKEN AS 0..(2**24)-1;
        END RECORD;
```




1.1.3.2.4 COMPLEX

COMPLEX data is represented in storage as an ordered pair of REAL values, addressed as a unit by addressing the low-address end of the low-address REAL value. The two values represent the real part and the imaginary part of a COMPLEX value, respectively.

The two values must be fetched separately and processed as two different REAL values. Each part is fetched left-justified, zero-filled, truncated on the right.




```
TYPE complex IS RECORD
        real_part:real;
        imag_part:real;
        END RECORD;
```

1.1.3.2.5 DOUBLE PRECISION

Double precision data occupies two 32-bit storage containers. No address alignment restrictions are enforced; any 32-bit container may be the start of a double precision value. The data representation is also "IBM format" (sign bit followed by a 7-bit, excess-64 hexadecimal exponent, followed by a 56-bit hexadecimal normalized mantissa).

Double precision values are addressed at the low- address end, left-justified, zero-filled on the right, truncated on the right.


```
TYPE double_precision IS BIT(64) TAKEN AS RECORD
    sign        :(0=positive, 1=negative)
    exponent    :BIT(7) TAKEN AS 0..127,
    mantissa    :BIT(56) TAKEN AS
                  0..(2**56)-1;
    END RECORD;
```

Type DOUBLE PRECISION differs from REAL in length only.

### 1.1.3.2.6 COMPLEX DOUBLE

COMPLEX DOUBLE data is represented as an ordered pair of DOUBLE PRECISION values, addressed from the low-address end of the low-address value. The values represent the real part and the imaginary part of a COMPLEX value, respectively. They are fetched separately, left-justified, zero-filled on the right, and treated as two DOUBLE PRECISION values.

```
TYPE complex_double IS RECORD
        real_part       :double precision;
        imaginary_part  :double precision;
        END RECORD;
```

### 1.1.3.2.7 LOGICAL

LOGICAL data is represented as a 32-bit containerized bit vector, of which only the first (low-address) bit is significant. The rest of the container is ignored. The FORTRAN standard requires that LOGICAL's be represented in this way.

LOGICAL data is addressed at the low-address end, right-justified and sign-filled, truncated on the right.

### 1.1.3.3 General Notes on Data Types

Length and fetch mode fields dictate alignment and padding upon data fetch and truncation upon data store. Significant bits of a floating point (REAL or DOUBLE PRECISION) value lost through truncation are ignored. Significant bits of an INTEGER value lost due to truncation are considered a fixed point overflow and must generate an exception trap.

Arithmetic instructions are typped only, and imply no containerization. Operand length is specified by the operand's associated NTE. The NTE also controls fetching and storing (i.e., dictates justification, fill, and truncation). By policy, a compiler will never generate an uncontainerized instruction with different operand sizes. Such an instruction is undefined within the FORTRAN machine.

Certain instructions do denote containerization. These instructions implement untyped bit moves, conversions between data representations, and/or containerization changes. The opcodes of such instructions are unique.

## 1.2 Execution Exceptions

The FORTRAN machine defines a number of execution exceptions. When an exception arises, several scenarios are possible. The first is to trap immediately with an appropriate error indication. The second is to totally ignore the exception and store a possibly incorrect result. A third scenario is to suppress the exception; i.e., store some approximate result and continue processing. In any case, a record of the exception is made in virtual machine state which may be tested (and cleared) by the program at a later time.

The following is a list of execution exceptions:

0 -- Exponent Overflow

> The result of a REAL or DOUBLE PRECISION computation has exceeded the range representable by the machine. If the exception is ignored, the exponent wraps around, so the resulting exponent is the the low order 7 bits of the proper result exponent. If the exception is suppressed, a result of infinity (i.e., the largest value representable) is stored with the proper algebraic sign.

1 -- Exponent Underflow

The result of a REAL or DOUBLE PRECISION computation is too small to be represented by the machine. If the exception is ignored, the exponent wraps around, so the resulting exponent is the low order 7 bits of the proper result exponent. If the exception is suppressed, a result of 1/infinity (i.e., the smallest nonzero value representable by the machine) is stored with the proper algebraic sign.

2 -- Floating Point Divide by Zero

A REAL or DOUBLE PRECISION division by zero has been attempted. This exception may not be ignored. If it is suppressed, a result of infinity (i.e., the largest value representable by the machine in floating point) is stored with the proper algebraic sign.

3 -- Floating Point Data Error

The value recieved as a floating point operand is either not properly normalized or is unnormalizable (e.g., a nonzero exponent and zero mantissa). If the exception is ignored, the value is normalized prior to attempting the operation (or a true zero is inserted). The exception may not be suppressed.

4 -- Illegal Operand Value

The argument of an operation is out of range; for example, EXP(x) where raising e to the x would result in an exponent overflow. If the exception is ignored, the operation is attempted with the specified value. Any subsequent exceptions are handled as they occur. This exception may not be suppressed.

5 -- Protection or Access Violation

No response other than trap is possible.

6 -- Fixed Point Overflow

A fixed point (INTEGER) result contains more significant bits than can fit in the specified container. Significant data bits (i.e., bits other than copies of the sign bit) must be truncated from the left end of the result.

The result is not stored.

This exception may not be ignored or suppressed.

7 -- INTEGER Conversion Error

The result of a FIX operation cannot fit in a 32 bit INTEGER container. No response other than trap is possible.

8 -- Fixed Point Divide by Zero

Integer divide by zero has been attempted. This exception may not be ignored or suppressed.

9 -- Illegal Intra-Procedure Object Address

An absolute name syllable resolved to an address which is not within the current procedure object. This exception may not be ignored or suppressed.

10 -- Size Error

The source operand in an assignment operation is too large to fit in the specified sink operand. If this exception is ignored, the source is truncated to fit in the sink container (on the right for floating point, on the left for INTEGER and CHARACTER).

## 1.3 Virtual Machines for FORTRAN S-Language

The FORTRAN S-Language standard defines two disjoint classes of storage: CHARACTER and classical. For the purpose of manipulating these data classes, we define two slightly overlapping virtual FORTRAN machines. The untyped MOVE instruction is shared by both virtual machines.

### 1.3.1 CHARACTER Machine

CHARACTER IF Operations

Moves are accomplished using the untyped move instruction.

* if_=string <name-a> <name-b> <relative offset>

character strings "a" and "b" are compared and, if equal, the relative branch is taken. Otherwise, the next sequential instruction is executed.

Exceptions: 5

* if_>string <name-a> <name-b> <relative offset>

character strings "a" and "b" are compared and, if string "a" is greater than string "b," the relative branch is taken. Note that the bytes are compared as unsigned, 8-bit values.

Exceptions: 5

* if_>=string <name-a> <name-b> <relative offset>

character strings "a" and "b" are compared and, if string "a" is greater than or equal to string "b," the relative branch is taken. Otherwise, the next sequential instruction is executed.

Exceptions: 5

Notes:

Strings are addressed from low-address end of left-most byte.

Strings are compared as unsigned, 8-bit values, space filled on the right if necessary to make the comparand strings equal in length.

Substring Operation

The substring operation is realized by treating the string as a character array, i.e. using the indexing facility to locate the start of the substring and using the untyped move to move the desired characters.

Concatenate Operation

The concatenate operation is simply an untyped move, coupled to an integer arithmetic operation of the "named" length if necessary. The operation on the length is included as a separate instruction in the I-stream.

* move_spac <name-a>

ASCII spaces are moved to the string operand "a." The number of spaces to be stored is determined by the length of "a." If <length of "a"> mod 8 is not zero, a size error exists, and the last character stored is truncated to fit in the source string.

Exceptions: 5, 10

1.3.2 Classical FORTRAN Machine

1.3.2.1 Unary Operations

* MOV <name-b> <name-a>

move value of "b" to variable "a." This is an untyped move. It is assumed that this instruction is used to move values between identical containers.

Exceptions: 5, 10 (containers not identical)

* MOVREAL <name-b> <name-a>

Floating point value "b" is moved to floating point variable "a." If necessary, "b" is zero extended on the right or truncated on the right to fit properly in "a's" container.

Exceptions: 3, 5

* MOVCMPLX <name-b> <name-a>

Both components of COMPLEX value "b" are zero extended or truncated to fit the containerization of COMPLEX variable "a." Each component replaces the corresponding component of "a".

Exceptions: 3, 5

The above instructions implement conversions between REAL and COMPLEX containerizations.

* IFIX <name-a> <name-i>

IFIX converts REAL or DOUBLE PRECISION value "a" to INTEGER (or INTEGER*2); the result replaces INTEGER variable "i."

Exceptions: 3, 5, 7

This instruction implements a REAL to INTEGER conversion. The source operand may be REAL or DOUBLE PRECISION; the sink operand may be INTEGER or INTEGER*2.

* FLOAT <name-i> <name-a>

INTEGER or INTEGER*2 value "i" is converted to REAL or DOUBLE PRECISION. The result replaces REAL or DOUBLE PRECISION variable "a."

Exceptions: 5

This instruction implements the conversion from INTEGER or INTEGER*2 to REAL or DOUBLE PRECISION. Lengths are implied by the operands' NTE's.

1.3.2.2 Pointer Instructions

The following instructions build and convert pointers. The FORTRAN standard does not define pointers, however, they are still needed to address parameters and resolve external calls.

* PTR <name-q> <name-p>

The pointer value "q" is moved to pointer variable "q." Pointer formats may be changed for efficiency.

Exceptions: 5

* ADDR <name-a> <name-p>

The logical address of "a" is moved to pointer "p". The format of the stored pointer is chosen for efficiency.

Exceptions: 5

Two pointer formats are useful within the FORTRAN machine: the UID pointer and the self-absolute pointer. Since self-absolute pointers are more efficient, they are used whenever possible.

A pointer is always converted to internal format upon fetch. If the source pointer was a self-absolute pointer, the internal UID of the pointer's logical address is inserted as the fetched pointer's internal UID. If the internal UID of this pointer is the same as that of the destination variable's logical address, the pointer is stored as a self-absolute pointer. Otherwise, the internal UID must be converted to a real UID and the pointer stored as a full UID pointer.

In pushing parameter pointers, the call instruction builds self-absolute pointers whenever the parameter resides in the stack object (when the parameter's logical address reflects the UID of the stack object, in which the parameter pointer will reside).

1.3.2.3 Data Conversion Instructions

* NEG <name-b> <name-a>

Compute 0.00 minus REAL or DOUBLE PRECISION value "b", the result replaces REAL or DOUBLE PRECISION variable "a".

Exceptions: 3, 5

* INEG <name-j> <name-i>

Compute the 2's complement of INTEGER or INTEGER*2 variable "j," the result replaces INTEGER or INTEGER*2 variable "i."

Exceptions: 5, 6

* ABS <name-b> <name-a>

Compute the absolute value of REAL or DOUBLE PRECISION value "b," the result replacing REAL or DOUBLE PRECISION variable "a."

Exceptions: 3, 5

* IABS <name-j> <name-i>

Compute the absolute value of INTEGER or INTEGER*2 value "j," the result replaces INTEGER or INTEGER*2 variable "i."

Exceptions: 5

1.3.2.4 ASSIGNMENT Instructions

In the following instructions, "real" means either REAL or DOUBLE PRECISION; "integer" means INTEGER or INTEGER*2. By policy, no compiler will generate an instruction with differing operand lengths. Such an instruction is undefined.

1.3.2.4.1 Two Address Real

* ADD2 <name-a> <name-b>

Add the real values of "a" and "b," the result replacing "a."

Exceptions: 0, 1, 3, 5

* SUB2 <name-a> <name-b>

Subtract the real values "a" and "b" (a minus b), the result replacing "a."

Exceptions: 0, 1, 3, 5

* MUL2 <name-a> <name-b>

multiply real values "a" and "b" (a times b), the result replacing "a."

Exceptions: 0, 1, 3, 5

* DIV2 <name-a> <name-b>

DIV2 divides real value "a" by real value "b," the result replacing "a."

Exceptions: 0, 1, 2, 3, 5

1.3.2.4.2 Two Address Integer

* IADD2 <name-i> <name-j>

Add integer values "i" and "j", the result replacing "i"

Exceptions: 5, 6

*  ISUB2 <name-i> <name-j>

Subtract integer value "j" from "i", the result replacing "i".

Exceptions: 5, 6

*  IMUL2 <name-i> <name-j>

Multiply integer values "i" and "j", the result replacing "i"

Exceptions: 5, 6

*  IDIV2 <name-i> <name-j>

Divide integer value "i" by integer value "j", result replacing "i"

Exceptions: 5, 6, 8

1.3.2.4.3 Two Address LOGICAL

*  NOT <name-b> <name-a>

logical complement of bit string value "b" replaces bit string variable "a"

Exceptions: 5

1.3.2.4.4 Three Address Real

* ADD3 <name-b> <name-c> <name-a>

add real values of "b" and "c", result replacing real variable "a"

Exceptions: 0, 1, 3, 5

* SUB3 <name-b> <name-c> <name-a>

subtract real values of "b" and "c", result replacing real variable "a"

Exceptions: 0, 1, 3, 5

* MUL3 <name-b> <name-c> <name-a>

multiply real values of "b" and "c", result replacing real variable "a"

Exceptions: 0, 1, 3, 5

* DIV3 <name-b> <name-c> <name-a>

divide real value of "b" by "c", result replacing "a"

Exceptions: 0, 1, 2, 3, 5

1.3.2.4.5 Three Address Complex

In the following instructions, each name operand refers to a pair of real or double precision values representing complex (either complex or complex double) values.

* CADD3 <name-b> <name-c> <name-a>

Complex values "b" and "c" are added, real part to real part, imaginary part to imaginary part, the result replacing complex variable "a"

Exceptions: 0, 1, 3, 5

* CSUB3 <name-b> <name-c> <name-a>

Complex values "b" and "c" are subtracted ("c" from "b"), real part from real part, imaginary part from imaginary part, the result replacing complex variable "a"

Exceptions: 0, 1, 3, 5

* CMUL3 <name-b> <name-c> <name-a>

Complex values "b" and "c" are multiplied, the result replacing complex variable "a".

Exceptions: 0, 1, 3, 5

Note: Complex multiplication proceeds as follows:
"b" = r + js "c" = t + ju
b*c = (r + js) * (t + ju)
= (r*t + r*ju) + (js*t + js*ju)
= rt-su + j*(ru + st)
where REAL (b) =r, IMAG (b) =s; REAL (c) =t IMAG (c) =u

* CDIV3 <name-b> <name-c> <name-a>

Complex values "b" and "c" are divided according to the rules for complex arithmetic, the result replaces complex variable "a".

Exceptions: 0, 1, 2, 3, 5

Note: The following is the definition of complex division:
given the operation a = b / c for a, b, and c
complex variables,

REAL (a) = (REAL(b)*REAL(c) + IMAG(b)*IMAG(c))
/ ( (REAL(c))**2 + (IMAG(c))**2 )

IMAG(a) = (IMAG(b)*REAL(c) - REAL(b)*IMAG(c))
/ ( (REAL(c))**2 + (IMAG(c))**2 )

1.3.2.4.6 Three Address Integer

* IADD3 <name-j> <name-k> <name-i>

* ISUB3

* IMUL3

Add, subtract, or multiply integer values "j" and "k" (j+k, j-k, or j*k), the result replacing integer variable "i".

Exceptions: 5, 6

* IMOD3

Divide integer values "j" by "k"; the remainder of this division replaces integer variable "i".

Exceptions: 5, 6, 8

*  IDIV3

divide integer variables "j" by "k", the quotient of this division replaces integer variable "i".

Exceptions:  5, 6, 8

1.3.2.4.7 Three Address LOGICAL

*  AND3 <name-j> <name-k> <name-i>

*  OR3

*  XOR3

Logical "and", "or", or "exclusive-or" bit string values of "j" and "k", result replacing bit string variable "i"

Exceptions:  5

1.3.2.5 IF Instructions

Two-comparand instructions specify three-operand syllables--two names followed by a relative branch syllable.

One-comparand instructions specify two-operand syllables--one name followed by a relative branch syllable.

Relative branch syllables are signed, nibble-granular values.

If the relation is satisfied, the current PC is updated by sign-extending the relative branch operand value, multiplying it by four, and adding it to the PC offset, effecting a branch that is relative to the start of the IF instruction. See "Relative Branches" in the above section titled "Operand Types for the FORTRAN S-Language."

1.3.2.5.1 Two Comparand IF's

The following operation codes operate on either real or integer data. Both comparands must be of the same type.

* IF= <name-a> <name-b> <relative offset>

The real or integer values of "a" and "b" are compared algebraically and, if they are equal, a relative branch is taken.

Exceptions: 5

* IF<>

The real or integer values of "a" and "b" are compared algebraically and, if they are not equal, a relative branch is taken.

Exceptions: 5

The following operation codes operate on a particular data type.

*  IF_a>b

The real values "a" and "b" are compared algebraically and, if a is greater than b, a relative branch is taken.

Exceptions: 5

*  IF_a>=b

The real values "a" and "b" are compared algebraically and, if a is greater than or equal to b, a relative branch is taken.

Exceptions: 5

*  IF_i>j

The integer values "i" and "j" are compared algebraically and, if i is greater than j, a relative branch is taken.

Exceptions: 5

*  IF_i>=j

The integer values "i" and "j" are compared algebraically and, if i is greater than or equal to j, a relative branch is taken.

Exceptions: 5

1.3.2.5.2 One Comparand IF's

The following operation codes operate on either real or integer data. The second comparand is implicitly a true zero.

* IF=0 <name-a> <relative offset>

The real or integer value of "a" is compared algebraically to a true zero and, if a equals zero, a relative branch is taken.

Exceptions: 5

* IF<>0

The real or integer value "a" is compared algebraically to zero and, if not equal, a relative branch is taken.

Exceptions: 5

* IF>=0

The real or integer value "a" is compared algebraically to zero and, if not less than, a relative branch is taken.

Exceptions: 5

*  IF<=0

The real or integer value "a" is compared algebraically to zero and, if not greater, a relative branch is taken.

Exceptions:  5

*  IF>0

The real or integer value "a" is compared algebraically to zero and, if greater than, a relative branch is taken.

Exceptions:  5

*  IF<0

The real or integer value "a" is compared algebraically to zero and, if less than, a relative branch is taken.

Exceptions:  5

1.3.2.6 GOTO instructions

* GO_REL <relative offset>

The current PC is updated by the relative branch syllable. See "Relative Branches" in the above section titled "Operand Types for the FORTRAN S-Language.)

Exceptions: 5

* GO_OFFSET <name-i>

The integer value "i" replaces the offset portion of the current PC, effecting an intra-procedure object branch.

Exceptions: 5

This instruction is used to implement ASSIGNed GO's and branches beyond the range of a relative syllable. A FORTRAN label is assigned to an INTEGER variable by moving an integral literal.

* GO_COMP <addr-count> <name-selector> <relative_offset1> ... <relative_offsetn>

This instruction is a computed GOTO. The first operand is a literal syllable denoting the number of branch addresses specified. The second operand is the name of an integer variable representing the selector value. One relative branch syllable follows for each branch address specified in the literal count operand.

Exceptions: 5

```
PROCEDURE GO_comp (addr_count   :literal16,
                   name_selector:name,
                   offset_list  :array[addr_count]
                                   of relative_offset);

IF 0 < eval( resolve( name_selector ) ) < addr_count
        THEN
        PC := PC + offset_list[ eval( reslove( name_selecto
        ELSE
        next_sequential_instruction;
        ENDIF;
END PROCEDURE GO_comp;
```

1.3.2.7 DO Loop Control

*  BCT <name-i> <relative offset>

The integer value of "i" is compared to zero and, if equal, the next sequential instruction is executed. Otherwise, i is decremented by 1, the result replacing i. Following this, a relative branch is taken.

Exceptions: 5, 6

It may be assumed that i is decremented and the branch is taken every time.

```
PROCEDURE bct (name-i       :name,
               relative_offset :literal16);

VARIABLE operand_ptr  :POINTER;
END VARIABLES;

operand_ptr := resolve( name-i );

IF operand_ptr@ = 0
        THEN next_sequential_instruction
        ELSE
        operand_ptr@ := operand_ptr@ - 1;
        PC := PC + relative_offset;
        END IF;
END PROCEDURE bct;
```

*  BXLE <name-i> <name-inc> <name-limit> <relative offset>

*  BXH

The integer values of "i" and "inc" are added, the result replacing i. The result is compared algebraically to integer value "limit". A relative branch is taken if the comparison yields less than or equal (BXLE) or greater than (BXH). This instruction functions identically to the 360 BXLE and BXH instructions.

Exceptions: 5, 6

```
PROCEDURE BXLE (name-i           :name,
                name-inc         :name,
                name-limit       :name,
                relative_offset :literal16);

resolve( name-i )@ := resolve( name-i )@ +
        resolve( name-inc )@;

IF resolve( name-i )@ <= resolve( name-limit)@
        THEN PC := PC + relative_offset
        ELSE next_sequential_instruction
        END IF;

END PROCEDURE BXLE;
```

*  BXLE1 <name-i> <name-limit> <relative offset>

The value of INTEGER variable "i" is incremented by 1, the result replacing i. If the result is less than of equal to INTEGER value "limit", the relative branch to "relative offset" is taken. Otherwise, the next sequential instruction is executed.

This instruction functions identically to "BXLE" with a an implicit increment of 1.

Exceptions: 5, 6

1.3.2.8 Miscellaneous Instructions

*  INC <name-i>

The integer value "i" is incremented by 1, the result replacing i.

Exceptions: 5, 6

*  DEC <name-i>

The integer value "i" is decremented by 1, the result replacing i.

Exceptions: 5, 6

*  ZERO <name-a>

Variable denoted by name "a" is cleared to a true zero. Note that this is an untyped operation.

Exceptions: 5

*  ONE <name-i>

Integer value "i" is set to a 2's complement value of one.

Exceptions: 5

*  ALLONES <name-i>

Integer or logical variable "i" is set to a 2's complement negative one (or the logical complement of zero, i.e. all ones).

Exceptions: 5

*  SET_RESPONSE <exception#> <scenario#>

This instruction informs the FORTRAN machine of the desired exception response. The first operand is a literal syllable representing an exception number (0 to 9). The second operand is a literal representing a response scenario (0 to 2 in the case of the FURIAN machine). Refer to the section on exceptions for a discussion of possible exceptions and responses.

Exceptions: 4 (illegal exception# or scenario#)

*  TEST_EXCEPTION <exception#> <name-i>

This instruction tests for the occurrence of an exceptional condition. The first operand is a literal denoting an exception #. The second operand is a name denoting an INTEGER variable. If the specified exception has occurred, the exception record is cleared and an integer value 1 is stored replacing "i". Otherwise, an integer value zero replaces "i".

Exceptions: 4 (illegal exception #), 5

1.3.2.9 CALL/RETURN Instructions

Three types of call instructions are included in the FURIAN S-Language. The first two implement internal calls; the third

implements an external call.

1.3.2.9.1 Internal CALL's

* PUSHJ <relative offset>

The offset of the current PC is pushed onto the stack and a relative branch is taken. Note that this instruction does not define a new stack frame.

Exceptions: 5

* POPJ

The 32-bits on the top of the stack (whether or not they represent a value) are assumed to be a unsigned integer. The offset of the current PC is replaced by this value. The POPJ instruction implements a return from subroutines entered by PUSHJ.

Exceptions: 5

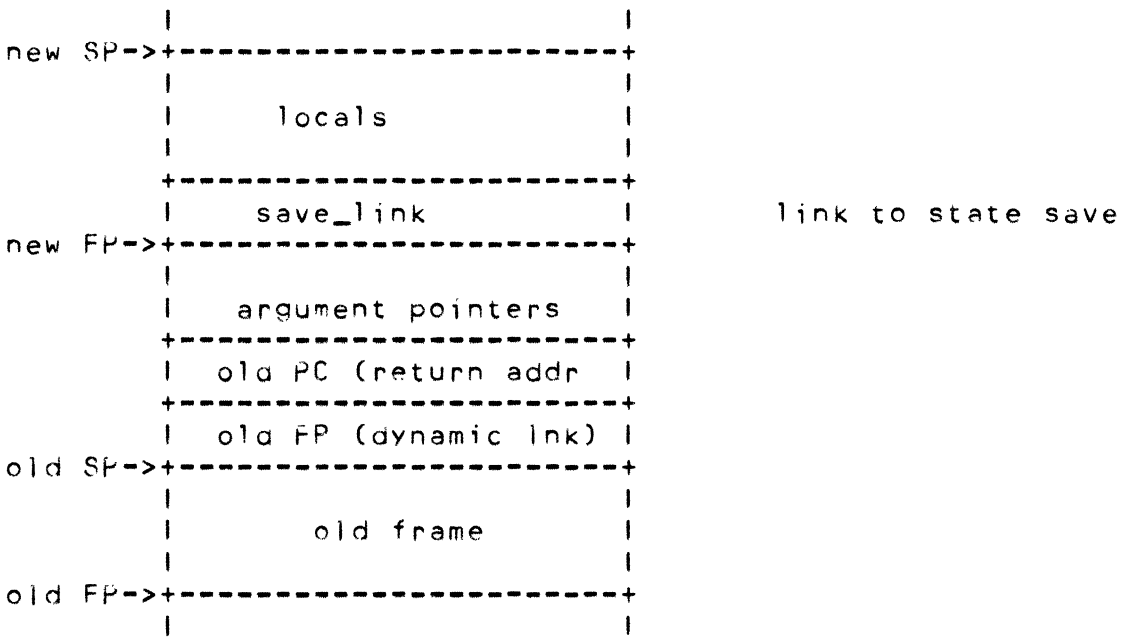
The above call is always intra-language. No entry headers are evaluated in such a call.

The actual format of the pushed PC offset is model dependent, since this activation record is created and interpreted dynamically.

1.3.2.9.2 Internal FORTRAN Calls

Since FORTRAN is a static language, it is unnecessary to save the entire state of a general external call. The following instructions implement an efficient intra-language, intra-procedure object call.

The stack is used to contain activation records (return address and old frame pointer), actual argument pointers, and locals (if necessary).




```
stack_frame IS RECORD
   state_save   :RECORD
        old_FP          :POINTER %to caller's frame
        old_PC          :POINTER %to return point
        END RECORD;

   argument_ptrs        :ARRAY[n] OF POINTERS;
   save_link    :POINTER TO state_save;
   locals          :BIT[ initial_frame_size ];

   END RECORD;
```

* CALLR <parm-count> <name-arg1>... <name-argn> <relative_offset>

* CALLA <parm-count> <name-arg1> ... <name-argn> <name-1>

This instruction implements a full internal FORTRAN call. Since an internal procedure utilizes the same Static Data Object and Stack Object as the calling procedure, only the FP offset and PC offset need be saved and restored upon return.

Since activation records are created and interpreted dynamically, state may be saved in any convient, model-dependent manner without impacting transportability of object programs or data.

The first operand is an unsigned literal specifying the number of actual parameters (arguments) being passed. One name syllable follows in the I-stream for each actual parameter. The last argument name is followed by an entry header address specified as a relative offset (CALLR) or an absolute offset (CALLA).

Exceptions: 5

```
PROCEDURE CALLR (arg_count        :literal16;
                 name-arg         :ARRAY[arg_count] OF
                                     names;
                 relative_offset  :literal16);

VARIABLE temp_reg: POINTER;
END VARIABLES;

temp_reg := SP;
PUSH(FP);
PUSH(PC);

% The PC is assumed to point to the NEXT SEQUENTIAL
% instruction.

FOR i:1..arg_count INCREASING   REPEAT
      PUSH( resolve( name_arg[i] );
      END FOR;

FP := SP;
PUSH( temp_reg);
SP := FP + (PC + relative_offset).initial_frame_size;
PC := (PC + relative_offset).code_pointer;

END PROCEDURE CALLR;
```

The absolute call (CALLA) differs from relative call (CALLR) in that the initial frame size and code pointer are located relative to POPTR by evaluating name-i, rather than relative to the current PC:

```
SP := FP + (POPTR+eval( resolve( name-i))).initial_frame_size;
PC := (POPTR + eval( resolve( name-i))).code_pointer;
```

*  RET

This instruction implements a return from a full internal FORTRAN call. It reestablishes the state which was modified by callr or calla.

Exceptions: 5

```
PROCEDURE RET  %no operands

SP := FP.save_link;    %old stack pointer is
                       %     link to state save area
PC := FP.save_link@.old_PC;
FP := FP.save_link@.old_FP;

END PROCEDURE RET;
```

1.3.2.9.3 External Call Instruction

*  CALLX <parm-count> <name-arg1> ... <name-argn> <name-entry>

The first operand is an unsigned literal syllable indicating the number of actual parameters. One name follows in the I-stream for each actual parameter. The last operand is a name which resolves to the address of a Gate pointer in some Procedure Object.

Exceptions: 5

* RETX

A return from an external call is performed.

Exceptions: 5

The external call is defined architecturally. See the product description for a discussion of its operation.

1.3.2.10 Extended Instructions

The following instructions are implemented in both single and double precision versions (REAL and DOUBLE PRECISION). "real" means REAL, and "double precision" means DOUBLE PRECISION.

1.3.2.10.1 Transcendental Functions

* SIN <name-b> <name-a>

The trigonometric SIN of real value "b" is calculated and replaces real variable "a"

Exceptions: 0, 1, 2, 3, 5

* COS

The trigonometric COS of real value "b" is calculated and replaces real variable "a"

Exceptions: 0, 1, 2, 3, 5

* TAN

The trigonometric TAN of real value "b" is calculated and replaces real variable "a"

Exceptions: 0, 1, 2, 3, 5

* ATAN

Calculate the arctangent of double precision value "b" and replace the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* ATAN2 <name-x> <name-y> <name-a>

Calculate the quotient of real value "y" divided by real value "x". If the calculation overflows, replace real variable "a" with the arctangent of "infinity". Otherwise, calculate the arctangent of "y"/"x" and replace "a"

Exceptions: 0, 1, 3, 5

* DSIN

Calculate the sin of double precision value "b" and replace double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DCOS

Calculate the cos of double precision value "b" and replace the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DTAN

Calculate the tangent of double precision value "b" and replace the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DATAN

Calculate the arctangent of double precision value "b" and replace the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DATAN2 <name-x> <name-y> <name-a>

Calculate the quotient of double precision values "y" divided by "x". If the calculation overflows, replace the double precision variable "a" with the arctan of "infinity". Otherwise, calculate the arctangent of "y"/"x" and replace "a"

| Exceptions: 0, 1, 3, 5

1.3.2.10.2 Hyperbolic Functions

* SINH

Calculate the hyperbolic sin of real value "b", the result replacing real variable "a"

| Exceptions: 0, 1, 2, 3, 5

* COSH

Calculate the hyperbolic cos of real value "b", the result replacing real variable "a"

| Exceptions: 0, 1, 2, 3, 5

* TANH

Calculate the hyperbolic tangent of real value "b", the result replacing real value "a"

| Exceptions: 0, 1, 2, 3, 5

* DSINH

Calculate the hyperbolic sin of double precision value "b", the result replacing double prec variable "a"

| Exceptions: 0, 1, 2, 3, 5

* DCOSH

Calculate the hyperbolic cos of double precision value "b", the result replacing double precision variable "a"

| Exceptions: 0, 1, 2, 3, 5

* DTANH

Calculate the hyperbolic tangent of double precision value "b", the result replacing double precision variable "a"

| Exceptions: 0, 1, 2, 3, 5

1.3.2.10.3 Other Intrinsic Functions

* EXP

The real value "e**<value of real variable "b"> is calculated and replaces real variable "a"

| Exceptions: 0, 1, 2, 3, 5

* LOG

The natural logarithm of real value "b" is calculated and replaces real variable "a"

| Exceptions: 0, 1, 2, 3, 5

* E10

The real value "10**<value of real variable "b"> is calculated and replaces real variable "a"

| Exceptions: 0, 1, 2, 3, 5

* LOG10

The common (base 10) logarithm of real value "b" is calculated and replaces real variable "a"

Exceptions: 0, 1, 2, 3, 5

* SQRT

Calculate the square root of the real value of "b" the result replacing the real variable "a"

Exceptions: 0, 1, 2, 3, 5

* DEXP

Calculate the natural logarithm base raised to the double precision value "b", the result replacing the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DLOG

Calculate the natural logarithm of the double precision value "b", the result replacing the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DE10

Calculate the common logarithm base (10.) raised to the double precision value "b", the result replacing double precision variable "a"

Exceptions: 0, 1, 2, 3, 5

* DLOG10

Calculate the common log. of double precision value "b" the result replacing double prec variable "a"

Exceptions: 0, 1, 2, 3, 5

* DSQRT

Calculate the square root of double precision value "b", the result replacing the double precision variable "a"

Exceptions: 0, 1, 2, 3, 5