```
+$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$+

                                   $      $   $$$$
                                   $      $   $   $
                                   $      $   $   $
                                   $      $   $   $
                                   $      $   $   $
                               $   $  $   $   $   $
                                $$$    $$$    $$$$

                        USER=JJD   QUEUE=LPT   DEVICE=@LPB1
               SEQ=4  QPRI=127  LPP=63  CPL=80  COPIES=1  LIMIT=76

                          CREATED:  24-OCT-77  16:09:00
                         ENQUEUED:   6-DEC-77  14:20:54
                         PRINTING:   6-DEC-77  14:27:26

                           PATH=:PDD:MEMO:MEMO$314.LS

   $   $  $$$$$  $   $   $$$     $    $$$$$    $       $             $       $$$
   $$ $$  $      $$ $$  $   $   $$$$     $    $$      $$             $      $   $
   $ $ $  $      $ $ $  $   $  $ $      $    $ $     $ $             $      $
   $ $ $  $$$$   $ $ $  $   $   $$$    $$$     $    $  $             $       $$$
   $   $  $      $   $  $   $     $ $     $    $    $$$$$            $          $
   $   $  $      $   $  $   $  $$$$   $   $    $       $     $$      $      $   $
   $   $  $$$$$  $   $   $$$     $     $$$   $$$$$     $     $$      $$$$$   $$$

+$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$+


                               AOS XLPT REV 01.00
```

To: S-Language Task Force

From: R. Belgard, D. Farber

Subject: Proposed SPL S-language definition

Date: 20/Oct/77 Memo No. 314

Abstract:

This document defines the data types, representations, and operations for an SPL s-language for the FHP system. The language is greatly simplified over previous proposals by the factoring out of all data addressing and type information into the namespace architecture.

1 Objectives of this Report

The intent of this document is to fully specify the formats, operations, and special conditions for an instruction set which can be used to efficiently implement normal SPL programs. A number of forseen uses of SPL will require or be aided by extensions to the S-language not specified here. These areas include

- Conversion, Editing, and Formatting
- Shared data mediation
- Other kernel microcode operations

All system calls are considered simply external calls into system-controlled domains.

In addition, it should be possible to add S-language operations in revisions of the S-language whenever it is found that such additions would significantly enhance the instruction set in some way. Such additions cannot reasonably be forseen, but can only be determined from dynamic statistics of real programs.

2 Data Types and Representations

The SPL s-language defines five types to represent all of SPL's data types and implicit structures. These data types are in general determined by the opcode; the type field of the NTE is currently ignored.

The instruction set assumes a "hidden register" model. In this model, the type field determines how operands are placed into arbitrarily large invisible registers - whether data is right or left adjusted, and if right adjusted, whether zero or sign filled. Operations then manipulate such registers, and finally store back results. A number of exception conditions occur when the result memory container is too small for the data stored in it.

The operation definitions listed determine how such data is interpreted (e.g. as integers, bits, etc.). In most cases this implies no type checking on the part of the interpreter, merely assumptions made.

2.1 Integers

Integers come in two forms, which are distinguished by their types in the name table and never by information within an s-language operator. Internally, all integers are represented identically as 16 bit signed quantities; all transformations of length and/or signedness occur in the fetch and store steps of an instruction.

2.1.1 Unsigned Integer

Unsigned integers represent SPL enumerations (including ASCII and BOOLEAN), and intervals, whenever the lower bound is non-negative. Additionally, unsigned integers will no doubt be used to represent the current lengths of strings.

An unsigned integer n bits in length may represent any value in the range 0 to 2**n-1. The maximum length of an unsigned integer is 63 bits (because the high order bit is used internally to store an implicit zero sign); the minimum length is 1 bit.

Unsigned integers are internally right adjusted and zero filled. All integer arithmetic and comparison operations interpret all integer values as signed. When stored, unsigned values are truncated on the left; if the truncated part is not all zeroes, an integer overflow trap is taken.

2.1.2 Signed Integer

Signed integers represent all SPL intervals where the lower bound is negative.

An unsigned integer n bits in length may represent any value in the range -2**(n-1) to 2**(n-1)-1. The maximum length for an unsigned integer is 64 bits; the minimum length is 1 bit.

Unsigned integers are internally right adjusted and sign-filled. When stored, unsigned values are truncated on the left; if the truncated part is not an extension of the sign bit (high order bit of the result) an integer overflow trap is taken.

2.2 Real

Real values are used to represent the SPL data type REAL. Real values are always 64 bits in length, and are represented in "IBM standard" form (8 bit radix 16 exponent and 48 bits of signed mantissa).

Real values are left adjusted and zero filled. When stored, the low order (rightmost) bits are truncated as necessary; this is a significance truncation which causes no trap.

2.3 Bit String

Bit strings are used to represent SPL sets. Additionally, bit string operations are sometimes used on boolean values and may be used on arrays and strings of boolean values.

Bit strings are internally left adjusted and zero filled. Bit strings may have a minimum length of 1 and a maximum length of 2**32-1 bits.

2.4 Pointers

Pointers are used for indirect addressing in the namespace architecture. They are used by the external CALL instruction, by the "STORE_POINTER" instructions, and by the "RESERVE" instruction which returns an address in the current frame.

Internally pointers are left adjusted and zero filled. They may be self relative, in which case only the upper 48 bits are significant, or absolute in which case all 128 bits are relevant. The SPL data type "pointer" is always represented by 128 bit absolute pointers; short forms may be used internally within the compiler.

The external CALL instruction always pushes a 128 bit pointer which may be absolute or self-relative. The RESERVE and STORE_POINTER instructions store any type of pointer; a trap is taken if the length is 48 bits and the pointer is not self relative. The STORE_FULL_POINTER instruction always stores a 128 bit absolute UID, even if it could be self-relative.

2.5 Typeless Data

A number of SPL operations deal with data merely as sequences of bits; these include many of the move and compare operations. The structured SPL data types - record, array, and string - may only be used in such "typeless" operations (and of course all field selection operations provided by the namespace architecture). The type field of operands in such operations is ignored; data is treated merely as bit strings of the length specified in the NTE. In this report we always call such data bit strings.

3 Operation Formats

The formats of operations for the S-language can be broken into three general categories: all operands are names, one operand is a self relative offset, and the first operand determines the total number of operands.

3.1 Operators whose operands are all names

Instructions of this format include all of the following forms of operation (where F and $F are the names assigned to S-language operations herein):

```
a := constant
a := F(a)
a := $F(b)
a := F(b,a)
a := $F(b,c)
```

Note that all operations starting with $ could easily be eliminated, by, for example, replacing the operation a:=$F(b,c) with the sequence a:=c; a:=F(b,a). However, using both format appears to be a natural optimization for these common cases.

Additionally the following special operations fall into this category.

```
GOTO n                     where n names a PO offset value
CALL p
CALL p(a)
RETURN
p := RESERVE(n)            reserves frame space
RELEASE(n)                 releases frame space
INTERNAL_CALL
INTERNAL_RETURN
STORE_POINTER
STORE_FULL_POINTER
SHIFT
```

3.2 Operators which include a relative program offset

Operations of this format comprise most control flow operations with the exception of those which specify a target address using a name instead of a self relative opcode. The operands n1 ... nj, if any, specify parameters of the predicate which determines whether or not to execute the branch instruction. The operand i specifies a signed, nibble offset relative to the beginning of the instruction, which is added to the PC if the branch is taken.

```
IF_F(a) goto i
IFNOT_F(a) goto i
IF_F(a,b) goto i
IFNOT_F(a,b) goto i
IF_F(a,b,c) goto i
IFNOT_F(a,b,c) goto i
```

Other operations involving PC offsets include:

LOOP
Comparisons with zero
FIND_FIRST_ONE
FIND_PREV_ONE
GOTO
LONG_GOTO

cause the iterand to be incremented by the increment value, and then branches if the iterand has passed the comparand. This instruction is used to implement most for-loops, as well as internally compiled loops.

3.3 Operators with a variable number of operands

The S-language contains two operations with a variable number of operands, the architecturally defined external call instruction and return_access instruction.

4 Special Conditions in the SPL S-language

This section deals with special conditions, especially trap conditions, which can be encountered during s-language execution. These conditions are as follows:

- Name resolution errors
- Instruction input and output errors
- Protection violations
- Control flow violations
- Long instructions
- Overlapping instructions

When an error is encountered during execution, a trap is taken. The exact way this is done will not be defined here. Traps may occur within instructions only for "long" instructions, which are so marked in this report.

4.1 Name Resolution Errors

Any instruction with at least one operand which is a name may generate a name resolution error. This includes all instructions except the GOTO instructions. Errors occur when the nametable or associated information is invalid or unreadable; these errors are described further in the namespace documents.

4.2 Instruction Input and Output Errors

Instruction input errors can occur when the type of an operand is not the type expected, or when the value of an operand is not within the legal range of values, or when the operand is larger than the maximum acceptable size. This category includes overflow, underflow, and divide by zero errors. In general, typechecking is not expected to be dynamically perforemd by the interpreter, however there is nothing to prevent it in the future. Length checking however will probably done, and range checking where necessary (as in string assignment) must be done to insure validity of the operation being performed. Output errors can be considered to mean "output container invalid for result". This condition may arise in at least three ways: when an integer value would be truncated on a store (this is an overflow or underflow), when in a memory to memory operation the destination field is not the same size as the source field(s), and when the type of the container is not valid for the result, in particular, when results are pointers.

4.3 Protection Violations

Whenever an instruction causes a fetch or store to main memory, a protection violation can arise. The details of this are specified elsewhere.

4.4 Control Flow Violations

Three sorts of control flow violation may arise. First, the offset of a local branch instruction can be out of bounds of the particular procedure/procedure object. (This sort of error may be subsumed by protection checks on instruction stream; but such checks do not guard against invalid jumps out of the current procedure but within the procedure object.)

The second sort of error occurs analogously when the offset for modification of the stack causes a stack to overflow or underflow, or when the size of a frame in reserve/release instructions is negative.

Finally, errors may be generated in the execution of call and return code. Such errors may arise via the kernel, or other s-language microcode, and hence are outside the scope of this paper. In general such errors may be described simply as "entry point specified is invalid", or "return packet on stack is invalid".

4.5 Long Instructions

A number of instructions in the SPL S-language may involve arbitrarily large amounts of data, and thus have some exceptional conditions. Primarily, such instructions must be interruptable both due to handling page faults and for handling external interrupts (which may of course involve process swapping.), and must be continuable after interrupt completion.

4.6 Overlapping Operands

It is important to note the effect of memory to memory instructions whose source and destination fields overlap. For simplicity and generality, we specify that the operation of any instruction, long or short, in which the destination field overlaps any source fields, is undefined, and may vary from model to model. The single exception to this is the string assignment instruction, which implements a bit shift and is guaranteed to work in any direction and for any length. Of course, those instructions in which a source field serves as a destination field, for example, add a to b, are not considered to be overlapping.

5 SPL Operation Definitions

The SPL operations are defined in detail below. The Operations are classified (arbitrarily) as control operations, and 1, 2, and 3-operand operators.

5.1 Definition Format

Each definition includes three sections.

The operands section defines the types expected for each operand. In the case where integers, reals, and bits are distinguished, type checking may be required by the interpreter to determine the required result type; in most cases the type of the operand can be assumed to be the type required. When the operand name given is "i", the operand is interpreted as a "k" bit signed value found in the i-stream, representing the number of nibbles from the start of the current instruction to the start of the destination instruction, if a branch is taken.

The semantics section defines the operational semantics of the opcode. The functions "resolve" and "evaluate" define the transformation of a name to the address it specifies, and a name to the value located at the address it specifies, respectively. Arithmetic operations such as + and *, and boolean operations such as logical-and and logical-complement, are assumed understood both for integer and real values.

Errors are specified by error code letters, which are fully defined in section 6.

5.2 Control Operations

5.2.1 GOTO i

```
        SEMANTICS:      PC <- PC + 4*i;
           ERRORS:      h
```

5.2.2 LONG_GOTO n1

n1:	integer
SEMANTICS:	PC <- PUPTR + evaluate(n1);
ERRORS:	b
NOTE:	allows "execution" of code by an interpreter

5.2.3 IF_ZERO n1, i

n1:	bit string, integer or real.
SEMANTICS:	IF evaluate(n1) = 0 THEN PC <- PC + 4*i,
ERRORS:	a,b,-

5.2.4 IFNOT_ZERO n1, i

n1:	bit string, integer, real
SEMANTICS:	IF evaluate(n1) <> 0 THEN PC <- PC + 4*i;
ERRORS:	a,b,-

5.2.5 IF_GT_ZERO n1, i

n1:	integer
SEMANTICS:	IF evaluate(n1) > 0 THEN PC <- PC + 4*i;
ERRORS:	a,b

5.2.6 IF_GE_ZERO n1, i

n1:	integer
SEMANTICS:	IF evaluate(n1) >= 0 THEN PC <- PC + 4*i;
ERRORS:	a,b

5.2.7 IF_LT_ZERO n1, i

```
        n1:      integer
 SEMANTICS:      IF evaluate(n1) <  0 THEN PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.8 IF_LE_ZERO n1, i

```
        n1:      integer
 SEMANTICS:      IF evaluate(n1) <= 0 THEN PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.9 REAL_IF_GT_ZERO n1, i

```
        n1:      real
 SEMANTICS:      IF evaluate(n1) >  0 THEN PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.10 REAL_IF_GE_ZERO n1, i

```
        n1:      real
 SEMANTICS:      IF evaluate(n1) >= 0 THEN PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.11 REAL_IF_LT_ZERO n1, i

```
        n1:      real
 SEMANTICS:      IF evaluate(n1) <  0 THEN PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.12 REAL_IF_LE_ZERO n1, i

```
        n1:      real
```

```
       SEMANTICS:      IF evaluate(n1) <= 0 THEN PC <- PC + 4*i;

          ERRORS:      a,b
```

5.2.13 IF_SUBSET n1, n2, i

```
          n1,n2:       bit string ( expected to be equal in length)
       SEMANTICS:      IF logical-AND (evaluate(n1) ,
                             logical-COMPLEMENT (evaluate(n2))) = 0
                                 THEN PC <- PC + 4*i;
          ERRORS:      a,b ,-
```

5.2.14 IFNOT_SUBSET n1, n2, i

```
          n1,n2:       bit string (expected to be equal in length)
       SEMANTICS:      IF logical-AND (evaluate(n1),
                            logical-COMPLEMENT (evaluate(n2)))
                               <> 0 THEN PC <- PC + 4*i;
          ERRORS:      a,b,-
```

5.2.15 IF_EQUAL n1, n2, i

```
          n1,n2:       bit strings of same length
       SEMANTICS:      IF evaluate(n1) = evaluate(n2) THEN
                               PC <- PC + 4*i;
          ERRORS:      a,b,-
```

5.2.16 IFNOT_EQUAL n1, n2 i

```
          n1,n2:       bit strings of same length
       SEMANTICS:      IF evaluate(n1) <> evaluate(n2) THEN
                               PC <- PC + 4*i;
          ERRORS:      a,b,-
```

5.2.17 IF_LESS n1, n2, i

```
     n1,n2:      bit strings
 SEMANTICS:      IF evaluate(n1) <  evaluate(n2) THEN
                         PC <- PC + 4*i;
    ERRORS:      a,b,-
      NOTE:      a < b if a and b are equal up to the end of
                 a and b is longer than a.
```

5.2.18 IFNOT_LESS n1, n2, i

```
     n1,n2:      bit strings
 SEMANTICS:      IF evaluate(n1) >= evaluate(n2) THEN
                         PC <- PC + 4*i;
    ERRORS:      a,b,-
```

5.2.19 INT_IF_EQUAL n1, n2, i

```
     n1,n2:      integer
 SEMANTICS:      IF evaluate(n1) = evaluate(n2) THEN
                         PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.20 INT_IFNOT_EQUAL n1, n2, i

```
     n1,n2:      integer
 SEMANTICS:      IF evaluate(n1) <> evaluate(n2) THEN
                         PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.21 INT_IF_LESS n1, n2, i

```
     n1,n2:      integer
 SEMANTICS:      IF evaluate(n1) < evaluate(n2) THEN
                         PC <- PC + 4*i;
    ERRORS:      a,b
```

5.2.22 INT_IFNOT_LESS n1, n2, i

```
        n1,n2:        integer
    SEMANTICS:        IF evaluate(n1) >= evaluate(n2) THEN
                              PC <- PC + 4*i;
       ERRORS:        a,b
```

5.2.23 REAL_IF_LESS n1, n2, i

```
        n1,n2:        real
    SEMANTICS:        IF evaluate(n1) <  evaluate(n2) THEN
                              PC <- PC + 4*i;
```

5.2.24 REAL_IFNOT_LESS n1, n2, i

```
        n1,n2:        real
    SEMANTICS:        IF evaluate(n1) >= evaluate(n2) THEN
                              PC <- PC + 4*i;
```

5.2.25 IF_INBOUNDS n1, n2, n3, i

```
     n1,n2,n3:        integers
    SEMANTICS:        IF (evaluate(n1) <= evaluate (n2)) AND
                         (evaluate(n2) <= evaluate(n3))  THEN
                             PC <- PC + 4*i
       ERRORS:        a,b
```

5.2.26 IFNOT_INBOUNDS n1, n2, n3, i

```
     n1,n2,n3:        integers
    SEMANTICS:        IF (evaluate(n1) > evaluate (n2)) OR
                         (evaluate(n2) > evaluate (n3)) THEN
                             PC <- PC + 4*i
       ERRORS:        a,b
```

5.2.27 FIND_FIRST_ONE n1, n2, i

```
             n1:      bits
             n2:      integer
      SEMANTICS:      temp <- evaluate(n2);
                      WHILE temp < length(n1) REPEAT
                              IF eval(resolve(n1)+temp) = "on" THEN
                                      resolve(n2) <- temp,
                                      RETURN;
                              END IF;
                              temp <- temp - 1;
                      END WHILE;
                      PC <- PC + 4*i;
         ERRORS:      a,b,o,-
```

5.2.28 FIND_PREV_ONE n1, n2, i

```
             n1:      bits
             n2:      integer
      SEMANTICS:      temp <- evaluate(n2);
                      WHILE temp >= 0 REPEAT
                              IF eval(resolve(n1)+temp) = "on" THEN
                                      resolve(n2) <- temp,
                                      RETURN;
                              END IF;
                              temp <- temp - 1;
                      END REPEAT;
                      PC <- PC+4*i;
          ERRORS:     a,b,o,-
```

5.2.29 LOOP n1, i

```
             n1:      integer
      SEMANTICS:      IF evaluate(n1) > 0 THEN
                              resolve(n1) <- evaluate(n1) - 1;
                              PC <- PC + 4*i;
         ERRORS:  a,b
```

5.2.30 CALL j, n0, n1, ... nj

Architecturally Defined External CALL Instruction

5.2.31 CALL0 n0

Architecturally defined External CALL Instruction
Restricted to 0 Parameters

5.2.32 CALL1 n0, n1

Architecturally Defined External CALL Instruction
Restricted to 1 Parameter

5.2.33 RETURN

Architecturally Defined External RETURN Instruction

5.2.34 RETURN_ACCESS j, n1, ... nj

Architecturally Defined External RETURN Instruction
with Dynamic Access Control Windows (DACs)

5.2.35 INTERNAL_CALL n1

```
           n1:      integer
    SEMANTICS:      SP@ <- PC offset of NEXT instruction: 32 bit i
                    SP <- SP + 32;
                    PC <- POPTR + evaluate(n1);
       ERRORS:      a,b,s
```

5.2.36 INTERNAL_RETURN

```
     SEMANTICS:      SP <- SP - 32;
                     PC <- PUPTR + fetch(SP,32 bit unsigned int)
        ERRORS:      a,b,s
```

5.2.37 RESERVE n1, n2

```
            n1:      integer
            n2:      pointer
     SEMANTICS:      IF evaluate(n1) < 0 then trap(range_error);
                     resolve(n2) <- create_pointer(SP);
                     SP <- SP + evaluate(n1);
        ERRORS:      a,s,i
```

5.2.38 RELEASE n1

```
            n1:      integer
     SEMANTICS:      IF evaluate(n1) < 0 then trap(range_error);
                     SP <- SP - evaluate(n1);
        ERRORS:      a,s,i
```

5.3 Single Name Instructions

5.3.1 CLEAR n1

```
             n1:      bit string
      SEMANTICS:      resolve(n1) <- zero_all_bits(length(n1));
         ERRORS:      a,-
```

5.3.2 SET n1

```
             n1:      bit string
      SEMANTICS:      resolve(n1) <- set_all_bits(length(n1));
         ERRORS:      a,-
```

5.3.3 SET_TO_ONE n1

```
             n1:      integer
      SEMANTICS:      resolve(n1) <- 1;
         ERRORS:      a
```

5.3.4 COMPLEMENT n1

```
             n1:      bit string
      SEMANTICS:      resolve(n1) <- logical-COMPLEMENT(n1);
         ERRORS:      a,-
```

5.3.5 NEGATE n1

```
             n1:      integer
      SEMANTICS:      resolve(n1) <- 0 - evaluate(n1);
         ERRORS:      a,o
```

5.3.6 ABSOLUTE_VALUE n1

```
          n1:      integer
   SEMANTICS:      resolve(n1) <- IF evaluate(n1) < 0 THEN
                             ( 0 - evaluate(n1)) ELSE evaluate(n1)
      ERRORS:      a
```

5.3.7 INCREMENT n1

```
          n1:      integer
   SEMANTICS:      resolve(n1) <- evaluate(n1) + 1;
      ERRORS:      a,o
```

5.3.8 DECREMENT n1

```
          n1:      integer
   SEMANTICS:      resolve(n1) <- evaluate(n1) - 1;
      ERRORS:      a,o
```

5.4 Two Name Instructions

In the following description of Binary Operations, unless otherwise noted, the types of the operands named by n1 and n2 is expected to be the same.

5.4.1 $COMPLEMENT n1, n2

```
     n1,n2:      bit strings
 SEMANTICS:      resolve(n2) <- logical-COMPLEMENT(n1);
    ERRORS:      a,o,-
```

5.4.2 $NEGATE n1, n2

```
     n1,n2:      integers
 SEMANTICS:      resolve(n2) <- 0 - evaluate(n1);
    ERRORS:      a,o
```

5.4.3 $ABSOLUTE_VALUE n1, n2

```
     n1,n2:      integers
 SEMANTICS:      IF evaluate(n1) < 0 THEN
                      resolve(n2) <- 0 - evaluate(n1);
                ELSE resolve(n2) <- evaluate(n1)),
    ERRORS:      a,o
```

5.4.4 $INCREMENT n1, n2

```
     n1,n2:      integer
 SEMANTICS:      resolve(n2) <- evaluate(n1) + 1;
    ERRORS:      a,o
```

5.4.5 $DECREMENT n1, n2

```
        n1,n2:      integer
    SEMANTICS:      resolve(n2) <- evaluate(n1) - 1;
       ERRORS:      a,o
```

5.4.6 REAL_NEGATE n1, n2

```
        n1,n2:      real
    SEMANTICS:      resolve(n2) <- 0 - evaluate(n1);
       ERRORS:      a
```

5.4.7 REAL_ABSVAL n1, n2

```
        n1,n2:      real
    SEMANTICS:      IF evaluate(n1) < 0 THEN
                            resolve(n2) <- 0 - evaluate(n1);
                    ELSE    resolve(n2) <- evaluate(n1);
       ERRORS:      a
```

5.4.8 MOVE n1, n2

```
       n1, n2:      any of same length
    SEMANTICS:      resolve(n2) <- evaluate(n1);
       ERRORS:      a,o,-
```

5.4.9 INT_MOVE n1, n2

```
       n1, n2:      integer
    SEMANTICS::     resolve(n2) <- evaluate(n1);
       ERRORS:      a,o
```

5.4.10 STORE_POINTER n1, n2

```
           n1:      any
           n2:      pointer
    SEMANTICS:      resolve(n2) <- create_pointer(resolve(n1));
       ERRORS:      a,o
```

5.4.11 STORE_FULL_PTR n1, n2

```
           n1:      any
           n2:      pointer
    SEMANTICS:      resolve(n2) <- create_full_ptr(resolve(n1));
       ERRORS:      a,o
         NOTE:      create_full_ptr differs from create_ptr -
                    it never stores object-relative pointers and
                    must have a 128 bit container for it.
```

5.4.12 SHIFT n1, n2

```
           n1:      integer
           n2:      bit string
    SEMANTICS:      resolve(n2) + n1 <- evaluate(n2)
       ERRORS:      a,-
         NOTE:      n1 may be negative for a left shift.
                    Full container is shifted, no fill is done;
                    this could be considered a "relative" move.
```

5.4.13 COUNT_ONES n1, n2

```
           n1:      bit string
           n2:      integer
    SEMANTICS:      resolve(n2) <- [ count of bits in evaluate(n1)
                                     which are "on"]
       ERRORS:      a,o,-
```

5.4.14 ROUND n1, n2

```
         n1:      real
         n2:      integer
  SEMANTICS:      temp <- evaluate(n1);
                  temp2 <- integer_part(temp);
                     IF temp > 0
                        IF fraction_part(temp) > 0.5 THEN
                                                    temp2 <- temp2 + 1;
                       ELSE
                        IF fraction_part(temp) > 0.5 THEN
                                                    temp2 <- temp2 - 1;
                  resolve(n2) <- temp2;
     ERRORS:  a,o
```

5.4.15 TRUNCATE n1, n2

```
         n1:      real
         n2:      integer
  SEMANTICS:      resolve(n2) <- integer_part(evaluate(n1));
     ERRORS:      a,o
       NOTE:      | integer_part(a) | = largest integer
                  less than or equal to |a|.
```

5.4.16 FLOAT n1, n2

```
         n1:      integer
         n2:      real
  SEMANTICS:      temp1 <- evaluate(n1);
                  temp2 <- 0;
                  resolve(n2) <- create_real(temp1,temp2);
     ERRORS:      a
```

5.4.17 AND n1, n2

```
      n1,n2:      bit string
  SEMANTICS:      resolve(n2) <- evaluate(n1) logical_AND
                                           evaluate(n2);
     ERRORS:      a,o,-
```

5.4.18 OR n1, n2

```
        n1,n2:      bit string
    SEMANTICS:      resolve(n2) <- evaluate(n1) logical_OR
                                       evaluate(n2);
       ERRORS:      a,o,-
```

5.4.19 AND_COMPLEMENT n1, n2

```
        n1,n2:      bit string
    SEMANTICS:      resolve(n2) <- evaluate(n1) logical_OR
                              logical-complement( evaluate(n2));
```

5.4.20 ADD n1, n2

```
        n1,n2:      integer
    SEMANTICS:      resolve(n2) <- evaluate(n1) + evaluate(n2);
       ERRORS:      a,o
```

5.4.21 SUBTRACT n1, n2

```
        n1,n2:      integer
    SEMANTICS:      resolve(n2) <- evaluate(n1) + evaluate(n2);
       ERRORS:      a,o
```

5.4.22 MULTIPLY n1, n2

```
        n1,n2:      integer
    SEMANTICS:      resolve(n2) <- evaluate(n1) * evaluate(n2);
       ERRORS:      a,o
```

5.4.23 QUOTIENT n1, n2

```
      n1,n2:      integer
  SEMANTICS:      resolve(n2) <- evaluate(n1) [integer divide]
                                   evaluate(n2);
     ERRORS:      a,o
       NOTE:      integer divide can be defined as
                  truncate(real divide).
```

5.4.24 REMAINDER n1, n2

```
      n1,n2:      integer
  SEMANTICS:      resolve(n2) <- evalaute(n2) -
                  ((evaluate(n2) [integer divide] evaluate(n1))
                        * evaluate(n1))
     ERRORS:      a,o
       NOTE:      sign of remainder is sign of dividend (n2).
```

5.5 Three Name Instructions

In the following instructions, unless otherwise noted, the types of the names in an instruction are expected to be the same. No type conversion is performed in the instruction execution.

5.5.1 $AND n1, n2, n3

```
      n1,n2,n3:      bit string
     SEMANTICS:      resolve(n3) <- evaluate(n1) logical_AND
                                        evaluate(n2)
        ERRORS:      a,o,-
```

5.5.2 $OR n1, n2, n3

```
      n1,n2,n3:      bit string
     SEMANTICS:      resolve(n3) <- evaluate(n1) logical_OR
                                        evaluate(n2)

        ERRORS:      a,o,-
```

5.5.3 $AND_COMPLEMENT n1, n2, n3

```
      n1,n2,n3:      bit string
     SEMANTICS:      resolve(n3) <- evalute(n1) logical_AND
                            logical_complement(evaluate(n2))
```

5.5.4 $ADD n1, n2 ,n3

```
      n1,n2,n3:      integer
     SEMANTICS:      resolve(n3) <- evaluate(n1) + evaluate(n2);
        ERRORS:      a,o
```

5.5.5 $SUBTRACT n1 ,n2, n3

```
      n1,n2,n3:      integer
     SEMANTICS:      resolve(n3) <- evaluate(n1) - evaluate(n2);
        ERRORS:      a,o
```

5.5.6 $MULTIPLY n1, n2, n3

```
      n1,n2,n3:      integer
     SEMANTICS:      resolve(n3) <- evaluate(n1) * evaluate(n2);
        ERRORS:      a,o
```

5.5.7 $QUOTIENT n1, n2, n3

```
      n1,n2,n3:      integer
     SEMANTICS:      resolve(n3) <- evaluate(n1) [integer divide]
                                    evaluate(n2);
        ERRORS:      a,o
```

5.5.8 $REMAINDER n1, n2, n3

```
      n1,n2,n3:      integer
     SEMANTICS:      resolve(n3) <- evaluate(n2) -
                     ((evaluate(n2) [integer divide] evaluate(n1))
                       * evaluate(n1));
        ERRORS:      a,o
```

5.5.9 REAL_ADD n1, n2, n3

```
      n1,n2,n3:      real
     SEMANTICS:      resolve(n3) <- evaluate(n1) + evaluate(n2);
        ERRORS:      a,o
```

5.5.10 REAL_SUBTRACT n1, n2, n3

n1,n2,n3:	real
SEMANTICS:	resolve(n3) <- evaluate(n1) - evaluate(n2);
ERRORS:	a,o

5.5.11 REAL_MULTIPLY n1, n2, n3

n1,n2,n3:	real
SEMANTICS:	resolve(n3) <- evaluate(n1) * evaluate(n2);
ERRORS:	a,o

5.5.12 REAL_DIVIDE n1, n2, n3

n1,n2,n3:	real
SEMANTICS:	resolve(n3) <- evaluate(n2) / evaluate(n1)
ERRORS:	a,o

6 Possible Future S-language Instructions

The flexibility of a microprogrammable machine is only effective when it can be used to delay decisions (binding, architecture, contracts) until sufficient information is available to determine just what instructions would be cost effective.

While we are convinced that the instruction set provided in the previous section will be adequate for generating code for arbitrary SPL programs, it may become clear that in certain cases the microprogramming of key routines, for instance, formatting routines or special table lookup routines, would provide definite performance improvements for a large class of programs.

The precise semantics of such routines should not be determined now. Rather, completed systems software should be measured carefully; those routines which turn out to be commonly used and which are bottlenecks can be microcoded. In order to help insure that programmers use common software to implement common functionality, and thus assure that we have a fair choice of commonly used interfaces to microprogram, it is necessary at this point to define and publicize libraries of low-level abstractions for use by FHP systems programmers. The SPL abstraction mechanism is ideal for this task.

Eventually, those abstractions which prove to be very commonly used and which could be significantly improved by microprogramming key routines, should be considered for such optimization. It is therefore important to ensure that small routines which can be programmed in the S-language can in general also be microprogrammed.

Below we present a number of example routines, written at SPL-slanguage level, which suggest themselves as candidates for microprogramming eventually.

6.1 SCAN_FOR_CHARACTER n1, n2, n3, i

```
           n1:      array of character
           n2:      character  (arbitrary size char)
           n3:      unsigned integer
            i:      pc offset
    SEMANTICS:      n3 := 0
                 a: if n1[n3] = n2 then return
                    increment n3
                    if n3 > length(n1) then goto i
                    goto a
```

6.2 SCAN_FOR_STRING n1, n2, n3, i

```
           n1:      string
           n2:      string (shorter than n)
           n3:      unsigned integer
            i:      offset
    SEMANTICS:      n3 := 0
                 a: if n1[n3:length(n2)] = n2 then return
                    increment n3
                    if n3 > length(n1) then goto i
                    goto a
```

6.3 TRANSLATE n1, n2

```
           n1:      array of character
           n2:      array of character
    SEMANTICS:      temp := length(n1)-1
                 a: n1[temp] := n2[n1[temp]]
                    if (temp := temp-1) >= 0 then goto a
```

6.4 HASH n1, n2

```
           n1:      string
           n2:      integer
    SEMANTICS:      temp := length(n1)
                 a: n2 := xor(n2,n1[temp])
                    if (temp := temp-1) >= 0 then goto a
```

6.5 SCAN_FOR_IN_SET n1, n2, n3, i

```
           n1:      string
           n2:      array of bits
           n3:      unsigned integer
            i:      offset
    SEMANTICS:      n3 := 0
                 a: if n2[ n1 [ n3 ] ] is "on" then return
                    increment n3
                    if n3 > length(n1) then goto i
                    goto a
```

6.6 CONVERT_INT_TO_STRING n1, n2

```
            n1:      integer
            n2:      string
     SEMANTICS:      converts n1 to right adjusted blank padded
                     character string n2.
```

6.7 ANALYZE_REAL n1, n2, n3

```
            n1:      real
            n2:      integer
            n3:      string
     SEMANTICS:      set n2 to base 10 exponent of n1
                     set n3 to mantissa of n1, a left adjusted
                     string with decimal implictly at left
```

6.8 FINITE_STATE_TRANSLATE n1, n2, n3

```
            n1:      finite state tables in some representation
                     of tuples <state, char, next state, output>
            n2:      string input
            n3:      string output
     SEMANTICS:      state := 1
                     index := 0
                a:   temp := output (state, n2[index])
                     state := next state (state, n2[index])
                     n3[index] := temp
                     if temp = 0 then return
                     increment index
                     goto a
```

6.9 FINITE_STATE_CHECK n1, n2, n3

```
            n1:      finite state tables as above
            n2:      string input
            n3:      integer output
     SEMANTICS:      state := 1
                     index := 0
                a:   temp := output (state, n2[index])
```

```
state := next state (state, n2[index])

n3 := state
if temp = 0 then return
increment index
goto a
```

7 Appendix: S-language Summary Charts

7.1 Operation, Operand Type, and Error Summary

Symbols in operands and errors columns have the following meanings:

operands		errors	
i	signed or unsigned int	a	name resolution error
r	real	o	result container size overflow
b	bits (any type as such)	s	stack violation (overflow etc.)
p	pointer	b	pc violation (out of bounds, etc.)
o	immediate offset	i	input value incorrect
x	p of external entry point	+	kernel generated error condition
k	immediate operand count	-	long instruction, interruptable
a	any type, not interpreted	all	statements with "a" error may generate fetch protection errors
*	operand is stored into	all	statements with "*" operands may generate store protection errors

operation	operands	errors
GOTO	o	b
LONG_GOTO	i	b
IF_ZERO	b o	ab-
IFNOT_ZERO	b o	ab-
IF_GT_ZERO	i o	ab
IF_GE_ZERO	i o	ab
IF_LT_ZERO	i o	ab
IF_LE_ZERO	i o	ab
REAL_IF_GT_ZERO	r o	ab
REAL_IF_GE_ZERO	r o	ab
REAL_IF_LT_ZERO	r o	ab
REAL_IF_LE_ZERO	r o	ab
IF_SUBSET	b b o	ab-
IFNOT_SUBSET	b b o	ab-
IF_EQUAL	b b o	ab-
IFNOT_EQUAL	b b o	ab-
IF_LESS	b b o	ab-
IFNOT_LESS	b b o	ab-
INT_IF_EQUAL	i i o	ab
INT_IFNOT_EQUAL	i i o	ab
INT_IF_LESS	i i o	ab
INT_IFNOT_LESS	i i o	ab

REAL_IF_LESS	r r o	ab
REAL_IFNOT_LESS	r r o	ab
IF_INBOUNDS	i i i o	ab
IFNOT_INBOUNDS	i i i o	ab
FIND_FIRST_ONE	b *i o	ab-
FIND_PREV_ONE	b *i o	ab-
LOOP	*i o	ab
CALL	k x a a ...	abs+
CALL0	x	abs+
CALL1	x a	abs+
RETURN		bs+
RETURN_ACCESS	k a a ...	abs+
INTERNAL_CALL	i	abs
INTERNAL_RETURN		bs
RESERVE	i *p	asi
RELEASE	i	asi
CLEAR	*b	a-
SET	*b	a-
SET_TO_ONE	*i	a
COMPLEMENT	*b	a-
NEGATE	*i	ao
ABSOLUTE_VALUE	*i	a
INCREMENT	*i	ao
DECREMENT	*i	ao
$COMPLEMENT	b *b	ao-
$NEGATE	i *i	ao
$ABSOLUTE_VALUE	i *i	ao
$INCREMENT	i *i	ao
$DECREMENT	i *i	ao
REAL_NEGATE	r *r	ao
REAL_ABSVAL	r *r	ao
MOVE	b *b	ao-
INT_MOVE	i *i	ao
STORE_POINTER	a *p	ao
STORE_FULL_PTR	a *p	ao
SHIFT	i *b	a-
COUNT_ONES	b *i	ao-
ROUND	r *i	ao
TRUNCATE	r *i	ao
FLOAT	i *r	ao
AND	b *b	ao-
OR	b *b	ao-
AND_COMPLEMENT	b *b	ao-
ADD	i *i	ao
SUBTRACT	i *i	ao

MULTIPLY	i *i	ao
QUOTIENT	i *i	ao
REMAINDER	i *i	ao
$AND	b b *b	ao-
$OR	b b *b	ao-
$AND_COMPLEMENT	b b *b	ao-
$ADD	i i *i	ao
$SUBTRACT	i i *i	ao
$MULTIPLY	i i *i	ao
$QUOTIENT	i i *i	ao
$REMAINDER	i i *i	ao
REAL_ADD	r r *r	ao
REAL_SUBTRACT	r r *r	ao
REAL_MULTIPLY	r r *r	ao
REAL_DIVIDE	r r *r	ao

7.2 SPL S-language Function Table

	Type of operand a		
Format of Function	Integer	REAL_	bits
IF_f(a) goto b	ZERO	ZERO	ZERO
	NOT_ZERO	NOT_ZERO	NOT_ZERO
	GT_ZERO	REAL_GT_ZERO	
	LT_ZERO	REAL_LT_ZERO	
	GE_ZERO	REAL_GE_ZERO	
	LE_ZERO	REAL_LE_ZERO	
IF_f(a,b) goto c	INT_EQUAL	EQUAL	EQUAL
IFNOT_f(a,b) goto c	INT_LESS	REAL_LESS	LESS
			SUBSET
IF_f(a,b,c) goto d	INBOUNDS		
IFNOT_f(a,b,c) goto d			
a := f()	CLEAR	CLEAR	CLEAR
	SET_TO_ONE		SET
a := f(a)	NEGATE		COMPLEMENT
b := $f(a)	ABSOLUTE_VALUE		
	INCREMENT		

	DECREMENT		
b := f(a)	FLOAT	ROUND	COUNT_ONES
		TRUNCATE	
		REAL_ABSVAL	
		REAL_NEGATE	
a := f(a,b)	ADD		AND
c := $f(a,b)	SUBTRACT		OR
	MULTIPLY		AND_COMPLEMENT
	QUOTIENT		
	REMAINDER		
c := f(a,b)		REAL_ADD	
		REAL_SUBTRACT	
		REAL_MULTIPLY	
		REAL_DIVIDE	

Miscellaneous Operations

GOTO	CALL	RETURN	MOVE
LONG_GOTO	CALL0	RETURN_ACCESS	INT_MOVE
LOOP	CALL1	INTERNAL_RETURN	STORE_POINTER
FIND_FIRST_ONE	INTERNAL_CALL	RESERVE	STORE_FULL_POINTE
FIND_PREV_ONE		RELEASE	SHIFT