APL - 145

ZZ20-6410

May 1971

PALO ALTO SCIENTIFIC CENTER

IBM
Data Processing Division

MPL/145 A LANGUAGE AND COMPILER FOR SYSTEM/370 MODEL 145 MICROPROGRAMMING

Daniel L. McNabb and John R. Walters, Jr.

Declassified

~~IBM INTERNAL USE~~

to add 32 bit wds:
W01 ← W01 |+| W24
W5(D0)→FDP ← W24
W01 ← W01 |+| W5(D0)→FDP

in 145: TABSET 2 22 31

to add:
WORK1 ← WORK1 |+| 4

W0 ← 0000
W1 ← 0004
W01 ← W01 |+| WORK1
WORK1 ← W01

WKSBAS + SAVEU

W0 = 0000
W1 = SAVEU
W01 = W01 |+| WKSBAS;

To STORE BYTE!

```
W0 ← ILR (H 0)
W0 ← W0 |A| 'FF00'
W2 ← W4(D3) → GPR3 (H0)
W2 ← W2 |A| '00FF'
W2 ← W2 |+| W0
W4(D3) → GPR3(H0) ← W2
```

NCIN p.20

p.29 count
The count Reg for RI is RV(By3)
RW is RU(By3)

Binary add p.20
1B } true add of bytes if S0=0
compl. add of bytes if S1=1

MPL/145

A

LANGUAGE AND COMPILER

FOR

SYSTEM/370 MODEL 145

MICROPROGRAMMING

p.11 { BX 03 (with T)
BH, bits 0-3 on branch { =0 if all are 0 / =1 otherwise
BL 4-7 { "

p.21 { XH D1, 0 result byte
XL 0, D0
X D1, D0
L 0, D1
H D0, 0

p.17 SHIFT

p.28 count register

WITH I24 only 24 bits are stored

WITH DD direct destination p.24
DS " source

DSC means bytes 31 ≡(H0)

S bit 0 true/complement
S bit 1 set to 1 for invalid decimal digit (decimal ops)
set to value of carry from bit 1 in binary (S12)
S bit 2 ∘ for byte ops: set to 1 if Z not 0 (S12)
∘ for word arith: set to 1 if Z(32 bits) not 0 (S12)
∘ for word arith: " " " Z(24 bits) " " (Z24)
∘ for storage word ops, set to 1 if count not 0 after decr (S2)
set to 0 if count is 0 " "
S bit 3 set to carry from bit 0
S bit 4,5 → byte ops: the byte
word ops (S12): byte 0
word ops (Z24): byte 1

S ← WORK2(3) + GO TO CEI(S4, B7)

"B" = present values
"S" = old values
∴ B7 means S Bit 7 after.. i.e. Bit 7 of WORK2(3)
(S4) means S Bit 4 before op.

Daniel L. McNabb
John R. Walters, Jr.

Palo Alto Scientific Center

with S12 { S1 = carry out of byte 0
S2 = 1 if result not zero, unchanged otherwise

with S45 { S4 = 1 if bits 0-3 of result are 0
S5 = 1 4-7 " " " "
set to 0 otherwise

byte arith p.21
word arith p.17

note: needs a pointer
W1 = '0002'; (odd W1,3 reg)

W3 = W1 → CMOD0

CMOD p.31

MMOD

GPR pointers must be even W0, 2, 4, 6
odd 1, 3, 5 (not 7)

CS

UPDATE p.30

(p.12)
0 = Z0 means S4 & S5 are 00
01
10
1 = Z0 means S4 & S5 are 11 i.e. byte is all 0

NZ p.12 means all 8 bits are tested
=0 if all are 0
=1 otherwise

Abstract

MPL/145 is an experimental language and compiler for microprogramming System/370 Model 145, employing compiler techniques, rather than assembler lore. The level of detail in the language is purposely kept high in order to enable the microprogrammer complete freedom in the use of the machine; therefore, MPL/145 may be termed either a machine-dependent compiler or a syntax-directed assembler. This report is intended as a reference manual for MPL/145 and is to be used in conjunction with the functional specifications of the System/370 Model 145.

Terms for IBM Subject Index

Microprogramming
Compilers
Programming Langauges
IBM System/370 Model 145

Table of Contents

Abstract page 2

Table of Contents page 3

I. Introduction page 4

II. Compiler Overview page 5

II.1 Examples page 5
II.2 Block Definition page 7
II.3 Declaration Statements page 8
II.4 Procedural Statements page 9

III. Language Description page 11

III.1.1 Branch Instruction page 11
III.1.2 Full-Word Arithmetic page 15
III.1.3 Byte Arithmetic page 18
III.1.4 Word Move page 24
III.1.5 Storage Reference page 26
III.1.6 CALL Statement page 32
III.1.7 RETURN Statement page 33

III.2 Declarations page 34
III.3 Special Control page 38

IV. Bibliography page 42

UPDATE p.30

I. INTRODUCTION

MPL/145 is a machine-dependent language for microprogramming System/370 Model 145. Generally speaking, MPL/145 statements have a one-to-one relation with Model 145 object micro-instructions, such that the detail of microprogramming is not subsumed in the language. This was, indeed, a key tenet in the design of the language, because it was considered absolutely essential for the microprogrammer to retain complete control over the hardware for whatever his purpose.

The MPL/145 langauge is derived from compiler technology, not assembler lore; the implementation employs an LR(k) syntax-directed scanning technique, and the design is highly parameterized for ease of maintenance and modification. Many of the features of the language have been borrowed from PL/I, ALGOL, and PL/360 (A). The basic form is that of PL/I; an MPL/145 program is a procedure, which may contain wholly nested procedures. This permits block structure and name scoping capabilities. A PL/I-like DECLARE statement is used as the principal way to name storage and constants peculiar to Model 145 microprogramming; however, there are no implicit naming rules, as there are in PL/I, so that all names must be declared explicitly. Finally, the computational statements in MPL/145, although essentially limited to the assignment statement, the GO TO statement, the CALL statement, and the RETURN statement, are patterned after their PL/I counterparts, the difference being that functions peculiar to Model 145 microprogramming have been added, while other functions, foreign to this machine, have been deliberately removed.

The remainder of this report presents the details of the MPL/145 language. It is assumed that anyone reading further for anything but a cursory glimpse of the language will be well acquainted with the micro-architecture of the Model 145, especially as specified in System/370 Model 145 Functional Specifications (B).

II. COMPILER OVERVIEW

II.1 MPL/145 Programming Example

The following example, courtesy of L. E. Lyon, contains an encompassing procedure, COMP, in which the seven variables appearing in the program are declared. Within this procedure are two other procedures, SUMM and FIBB, which find the sum from 1 to N and the N-th Fibonacci number, respectively. These latter procedures are called from a testing procedure, written as a separate procedure, TEST.

```
 1 COMP: PROCEDURE;
       /* DECLARATIONS FOR FIBB AND SUMM */
 2     DECLARE
           S    BYTE SYN(ES04(0)),ANS2 WORD SYN(LS14),
           RTN  WORD SYN(LS15),   ANS  WORD SYN(LS16),
           ANS1 WORD SYN(LS16),   NUM  WORD SYN(LS17),
           NUMB WORD SYN(LS17);
       /* FIND THE N-TH FIBONACCI NUMBER */
 3     FIBB: PROCEDURE;
 4         BGN:    ANS1 <- 1; ANS2 <- 1;
 6         TST:    NUMB <- NUMB |-| 2; GO TO MOR(NUMB(0:0));
 8         MOR(0): ANS1 <- ANS1 |+| ANS2;
 9                 ANS2 <- ANS2 |+| ANS1 & GO TO TST;
10         MOR(1): GO TO FIX(NUMB(3:7));
11         FIX(0): NUMB <- ANS1 & GO TO EXIT;
12         FIX(1): NUMB <- ANS2;
13         EXIT:   RTN <- RTN |+| 4;
14                 RETURN USING RTN;
15     END FIBB;
       /* FIND THE SUM FROM 1 TO N */
16     SUMM: PROCEDURE;
17            INN:       ANS <- NUM WITH DS;
18            MOR(1,1):  RESET S BY '20'X;
19                       NUM <- NUM |-| 1 WITH S12;
20                       ANS <- ANS |+| NUM & GO TO MOR(S2,S3);
21            MOR(0,1):  RTN <- RTN |+| 4;
22            OUT:       RETURN USING RTN;
23     END SUMM;
24 END COMP;
```

```
 1 TEST:PROCEDURE;
       /* PROCEDURE TO TEST FIBB AND SUMM */
 2      DECLARE BALR WORD SYN(LS15), COUNT WORD SYN(LS10),
            ARGMT WORD SYN(LS17);
 3      START:       GROUP(DLS2); COUNT <- 7;
 4      ALIGN(1,0): COUNT <- COUNT |-| 2; GO TO DONE(COUNT(0:0));
 6      DONE(0):     ARGMT <- COUNT WITH DS;
 7      ALIGN(0,0): CALL FIBB.BCN USING BALR;
 8      ALIGN(0,1): CALL SUMM.INN USING BALR;
 9      DONE(1):     BALR <- BALR WITH DS,STOP & GO TO DONE(1);
10 END TEST;
```

Syntactically, these procedures resemble PL/I quite closely. A free form is used, so that one or more statements may occupy a line, and comments, enclosed by '/*' and '*/', may appear within or between statements. Blanks are significant, only in that they may be used as well as other punctuation to terminate identifiers, constants, etc.

MPL/145 uses a '<-' to denote assignment in place of '=' used in PL/I. Because the micro-operations of the Model 145 have a very specific denotation, operations in MPL/145 are enclosed in '|', as seen in the operations |+| and |-| on lines 6, 8, 9, 13, 19, 20, and 21 of COMP, and line 4 of TEST. Hexadecimal constants are enclosed in quotes, followed immediately by the letter X.

In microprogramming the Model 145 it is possible to specify numerous latches, bits, switches, etc., ancillary to the main instruction, thereby providing many variants to a single micro-instruction. This is achieved in MPL/145 by using an optional WITH field, as seen in line 19 of COMP and line 6 of TEST. A branching capability is also available in conjunction with most micro-operations, and the optional & GO TO field appearing in lines 9, 11, and 20 of COMP, and in line 9 of TEST, is typical of the use of this facility.

Finally, the Model 145 hardware is built to handle indexed branches rather efficiently; and so MPL/145 permits the specification of indexed labels and their use in GO TO statements, as can be seen in many places in the above examples.

II.2 Block Definition (PROCEDURE/END)

An MPL/145 program must itself be a block which may contain other blocks nested to any arbitrary depth. All the declarative statements in a given block must appear before the procedural statements in that block, and all the procedural statements must appear before the beginning of an inner block, i.e., contained blocks must appear at the end of containing blocks. This is seen in the COMP example above.

A block is headed by a PROCEDURE statement of the form:

```
<label> : PROCEDURE;
```

The <label> names the procedure and becomes the name of the code module; it should consist of a letter followed by 3 alphanumeric characters. The PROCEDURE statement is used to establish the scope of variable names used in the program. By this means, a variable name may be defined in an inner block to have entirely different attributes from those associated with the same name in an outer or disjoint block. Thus the block structure may be used to build programs from independently produced pieces. The END statement is used to terminate a block and may have either the form:

```
END;
or,
END <label>;
```

In the latter case, the <label> corresponds to the <label> of a procedure. By using the <label> of an outer procedure, all procedures nested within the outer procedure may be terminated with a single END <label>; statement.

II.2.1 Labels

A procedural statement may have a label consisting of up to 12 alphanumeric characters, the first character of which must be a letter. For this implementation all statement labels are truncated to 6 characters.

```
<label>(<bx>,<bh>,<bl>):
```

The <label> may have up to three subscript fields specifying the particular leg of a branch-set. These are termed the <bx>, <bh>, and <bl> fields, respectively, and permissible values for them are:

```
<bx>
0-15

<bh>
0,1,X

<bl>
0,1,X
```

An X signifies that there is no leg corresponding to this subscript; leading X's or trailing blanks are not allowed, as the following examples indicate:

```
NAMEA(1)            valid subscript for <bl>
NAMEB(1,X)          valid subscript for <bh> and <bl>
NAMEC('D'X,0,1)     valid subscript for <bx>, <bh>, and <bl>
NAMED('E'X)         invalid subscript, should be NAMED('E'X,X,X)
NAMEF(3,1)          invalid subscript, should be NAMEF(3,1,X)
```

II.3 Declarative Statements (DECLARE/DCL)

All variables appearing in a program must be declared explicitly in a DECLARE statement appearing at the head of a procedure. The DECLARE statement is patterned after the PL/I DECLARE statement and is of the form:

```
DECLARE <variable definition list> ;
```

where <variable definition list> is a list of one or more variable definitions, each of which contains the name of the variable and a list of attributes. The statement may begin with either DECLARE or DCL.

Attributes are:

Data type attributes
BIT, BYTE, HALF, WORD

Storage class attributes
MAIN and CONTROL

Special attributes
SYN, BASED, EQU, OFFSET, and LINK

Further information can be found in section III.C.

II.4 Procedural Statements

There are seven instruction types in the Model 145:

- Branch Instruction
- Full-word Arithmetic Instruction
- Byte Arithmetic Instruction
- Word-move Instruction
- Storage Reference Instruction
- Call Instruction
- Return Instruction

In MPL/145 all but the Call and Return statements have the general form:

```
          <operation>  <with-field>  <branch control>
```

The <operation> for each statement type is different and is described in section III.

The <with-field> may optionally follow the <operation> and usually contains keywords to specify modifiers, masks, and status specifications. The <with-field> is written:

```
          WITH  <keyword>,<keyword>,...,<keyword>
```

where the <keyword> list, separated by commas, need not be in any specific order.

The <branch control> is used to specify the label of the next instruction to be executed, and may be written in any of the following forms:

```
          & GO TO <label>(<bx>,<bh>,<bl>)
            & GO TO <label>(<bh>,<bl>)
               & GO TO <label>(<bl>)
                  & GO TO <label>
```

It must follow the <with-field>, if there is one. In the absence of the <branch control>, the statement following is assumed to be executed next.

The <label> in each example may be any label in the program. The block structure is not used to limit the scope of a label. If a label is outside the current procedure, it must be qualified by a procedure name:

<procedure name>.<label>

The <bx>, <bh>, and <bl> fields are special identifiers or constants used to specify a particular leg of a branch-set. Permissible settings for these fields vary according to each instruction type; therefore, they are described in conjunction with the instructions in section III.

III. LANGUAGE DESCRIPTION

III.1.1 Branch Instruction

The general form of the Branch Instruction is:

```
<SET or RESET Function> & GO TO <label>(<bx>,<bh>,<bl>);
```

The branch instruction provides four functions:

1. Branching to another label in the same module.
2. Module switching.
3. Setting or resetting bits 0, 1, 2, 3, 4, 6, or 8 in a specified byte located in either Local or External Storage.
4. Setting or resetting certain selector channel circuit conditions.
5. Loading the S, T, or L-Registers.

Branching within a current module is permitted in combination with any of the functions, except module switching.

III.1.1.1 <bx>, <bh>, <bl> Subscript Fields

The label in the Branch Instruction may have up to three subscript fields, <bx>, <bh>, and <bl>, respectively. The values of these fields are used to determine which leg of the branch-set will be taken. The number of subscripts must correspond to the number of legs associated with the particular branch-set. The label may be written in any of the following ways:

```
<label>(<bx>,<bh>,<bl>)
 <label>(<bh>,<bl>)
   <label>(<bl>)
      <label>
```

Permissible <bx>, <bh>, and <bl> field subscripts are:

Absolute Values(May be Mnemonic)

<bx>	<bh>	<bl>
0-3	X,0,1	X,0,1

Values Obtained from S-Register Settings

<bh>	<bl>
S0,S1,S2,S4,S6	S3,S5,S7

Special identifiers S0 through S7 designate the bit in the S-Register used for the <bh> or <bl> subscript.

Branch-Source Bit Conditions

<bh>	<bl>
B0-B7,BH	B0-B7,BL,NZ,Z0

Special identifiers B0 through B7 specify the bit in the branch-source byte to be tested (see section III.1.1.2).

The special identifier, BH, appearing as the <bh> subscript is used to specify the first four bits, 0-3, of the branch-source byte. If all four bits are 0, the value of the subscript field is 0; otherwise the value is 1.

BH = 0 if all 0
=1 if any not 0

The special identifier, BL, appearing as the <bl> subscript has a similar effect as BH, except that the last four bits, 4-7, of the branch-source byte are tested. Again, the value of the <bl> subscript is 0 if all four bits are 0, and 1 otherwise.

The special identifier, Z0, is used as the <bl> subscript to specify bits 4 and 5 of the S-Register. If both these bits are 1, the value 1 is used for the subscript; otherwise the value 0 is used.

Z0

The special identifier, NZ, appearing as the <bl> subscript is used to test the entire branch source byte for 0's. If all 8 bits are 0, a 0 is used for the <bl> subscript; otherwise a 1 is used.

NZ

Using TH for the <bx> Subscript

The special identifier, TH, may be used as the <bx> subscript to designate bits 0 and 1 of the T-Register. This subscript is not permitted in conjunction with set or reset operations.

TH ≡ T_0, T_1

III.1.1.2 Branch Source

The branch-source byte may be a Local or External storage location. In addition to testing selected bits, as outlined above, they may be set or reset in the same instruction after being tested. Note that the testing of any bits in the branch-source byte is independent of any setting or resetting which might follow.

```
                    GO TO LEV(A(2:7),A(2:BL));
```

In this example the identifier A is a word-source with bit 7 of byte 2 providing the <bh> subscript and the low-order four bits, bits 4-7, specifying the <bl> subscript. The resulting 2 bits provide a four-way branch to the branch-set LEV.

Another form of the branch statement is:

```
   <s>
   <t> <- <branch-source> & GO TO <label>(<bx>,<bh>,<bl>);
   <l>
```

where <s>, <t>, and <l> are symbolic names for the S, T, and L-Registers. For example,

```
                 SREG <- A(2) & GO TO LEV(B7,BL);
```

In this example bits 4-7 of byte 2 of A are tested for zero. If all four bits are zero, the <bl> subscript is zero; otherwise the <bl> subscript is one. Bit 7 of byte 2 of A is used for the <bh> subscript.

The target register may only be: (1) the S-Register, (2) the T-Register, or (3) the L-Register, or an identifier which has one of these registers as its synonym. This does not prevent the user from using many names with these registers as synonyms.

III.1.1.3 Branch Instruction SET or RESET Function

The general form of the SET or RESET function is:

```
            <a>
 SET        <s>
 RESET      <p>    BY <constant> & GO TO <label>(<bx>,<bh>,<bl>);
             GA
```

where:

<a> names a byte in Local or External storage,

<s> is a name for the S-Register,

<p> is a name for the P-Register.

The Branch Instruction may be used to set or reset bits in the following various registers:

1. A Local or External storage byte also specified as the branch-source ,
2. The S-Register,
3. The P-Register,
4. The CA selector-channel circuit.

A branch-source byte may be used in the subscript field when setting or resetting the S or P-Registers, or the selector channel functions. The <constant> may be as follows:

'0h'x, 'h0'x, or 'hh'x,
where h is a hexadecimal digit from 0 to F

The SET function operates as follows:
wherever there is a 1 in the mask, the corresponding bit in the register is set to 1; all other bits remain unchanged.

The RESET function is similar in that:
wherever there is a 1 in the mask, the corresponding bit in the register byte is reset to 0; all other bits remain unchanged.

The selector-channel circuits can be set or reset using the special identifier, GA. Further information on this can be obtained in System/370 Model 145 Functional Specifications (D).

III.1.2 Full-Word Arithmetic

The general form of full-word arithmetic instructions is:

```
        <operation> WITH <with-field keywords>
          & GO TO <label>(<bx>,<bh>,<bl>);
```

III.1.2.1 Operations

The <operation> portion of the full-word arithmetic statement may have one of the following forms:

```
     1.  <a> <- <a> <op> <b>
     2.  <a> <- <a> <op> <constant>
     3.  <a> <- <a> <op> ( <constant expression> )
     4.  <a> <- <b>
     5.  <a> <- -<b>
     6.  <a> <- <constant>
     7.  <a> <- ( <constant expression> )
     8.  <a> <- -<constant>
     9.  <a> <- -( <constant expression> )
    10. <b> <- <a> <op> <b>
    11. <b> <- -<b>
    12.  ZW <- <a> <op> <b>
    13.  ZW <- <a>
    14.  ZW <- -<b>
```

In which:

    <a> specifies Local or External storage,
    <b> may only specify Local storage, and
    ZW   is a special identifier indicating no destination.

The <constant> can be a positive integer or symbolic constant less than 256. A <constant expression> must be enclosed in parenthesis.

The <op> refers to the following full-word operations:

1. |+| true add; <a> and <b> are added, and S0 is set to 0.

2. |-| two's complement add; 1 plus the complement of <b> is added to <a>, and S0 is set to 1.

3. |B| binary add; bit S3 is added to the value of <a> and <b> or its complement, depending on whether the value of S0 is 0 or 1, respectively.

When a specific <op> does not appear, as indicated on lines 4, 6, 7, and 13 above, the operation actually produced is:

0 |+| <b>

The minus sign, appearing on lines 4, 8, 9, 11, and 14 above, actually produces:

0 |-| <b>

Note: S0 is set to 1 as a result of this operation.

III.1.2.2 Partial <a> and <b> Source Inputs

A complete full-word for <a> or <b> need not be used in every operation. A word consisting of 16 low-order bits of an <a> source, such as ALPHA, may be indicated by writing:

ALPHA(16)

This is the only option for the <a> source other than the full-word.

Three options are permitted for specifying a word containing low-order bits from a <b> source, such as BETA:

1. BETA(4) four low-order bits.
2. BETA(8) low-order byte.
3. BETA(12) twelve low-order bits.

The high-order bits of the word are set to 0, unless the operation is either |-| or |B| with S0 set to 1, in which case the rest of the word is set to 1's.

III.1.2.3 Status Setting Specifications

The <with-field keywords> permitted in conjunction with full-word arithmetic statements are:

used in 3900 test

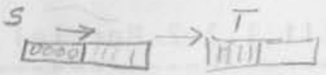
1. S12 S1 is set to the value of the carry-out of byte 0, bit 1 of the operation. S2 is set to 1 if the result is not 0; otherwise it is unchanged.

2. I24 Specifies that only 24 bits of the result are to be stored. No S-Register bits may be set or reset in conjunction with this option.

3. Z24 S2 is set to 1 if the low-order 24 bits of the result are not 0; otherwise it is unchanged. Only the low-order 24 bits of the result are retained.

III.1.2.4 Shifting

Shifting may also be specified using one of the following <with-field keywords>:



1. SHIFT logical right shift
2. SHIFTA arithmetic right shift
3. SHIFTTH shift right using bits T0-3

In the logical shift, the high-order four bits are set to 0's, while in the arithmetic shift the high-order four bits are set to the value of S0. SHIFTTH places the contents of bits 0-3 of the T-Register, T0-3, into the high-order four bits of the result.

In all cases, the four bits shifted out of the low-order positions are placed in T0-3.

Shifting is performed only on the <b> source, and only before the operation begins. When SHIFTA is specified, the value of S0 can not be set or reset, even though it may be specified in the operation.

III.1.2.5 Branching

A limited branching capability is possible with full-word arithmetic instructions; only three settings are permitted for the <bh> and <bl> subscripts, while <bx> may have four values:

<bx>	<bh>,<bl>
0-3	S2,S3
	S4,S5
	S6,S7

The corresponding bits in the S-Register are tested and used for branching, as explained above.

III.1.3 Byte Arithmetic

The general form of the byte arithmetic instruction is:

```
        <operation> WITH <with-field keywords>
                & GO TO <label>(<bx>,<bh>,<bl>);
```

III.1.3.1 Operations

The <operation> portion of the byte-arithmetic instruction may be any of the following:

```
   1.  <a>  <-  <a>  <op>  <b>
   2.  <a>  <-  <a>  <op>  <constant>       *
   3.  <a>  <-  <a>  <op> ( <constant expression> )  *
   4.  <a>  <-  <b>
   5.  <a>  <-  -<b>
   6.  <a>  <-  <constant>
   7.  <a>  <-  ( <constant expression> )
   8.  <b>  <-  <a>  <op>  <b>
   9.  <b>  <-  -<b>
   10.  Z   <-  <a>  <op>  <b>
   11.  Z   <-  <a>  <op>  <constant>       **
   12.  Z   <-  <a>  <op> ( <constant expression> )     **
   13.  Z   <-  <b>
   14.  Z   <-  -<b>
   15. <l>  <-  <a>  <op>  <b>      ***

   *   Operations |-| and |N| are not allowed with this form.
   **  Only operations |-| and |N| are allowed for this form.
   *** Only operations |X| and |+| WITH COUT are allowed.
```

<a> specifies a byte in Local or External storage, and <b> specifies a byte in Local storage. They can be written as either:

1. a byte-source identifier, or
2. a word-source identifier subscripted with 0, 1, 2, or 3 to specify a particular byte, or subscripted with the special identifier, I, to denote indirect byte addressing.

Examples are as follows:

byte-source identifier,
CBYTE

subscripted word-source identifier,
CWORD(0), CWORD(1), ..., CWORD(I)

A byte-source or subscripted word-source may also have gating information associated with the subscript. Examples of this are:

CBYTE(<gating control>)
or
CWORD(1:<gating control>)

The result may be placed in the L-Register by designating an identifier with the L-Register as its synonym, as shown on line 15, above.

The special identifier, Z, specified as the destination, is used to indicate that the result of the operation is not to be saved.

The <constant> may be a positive integer or symbolic constant less than 256. A <constant expression> must be enclosed in parenthesis.

III.1.3.2 Operations and Associated <with-field keywords>

The <op> field shown in the forms above is an actual micro-instruction, which may be modified using the following <with-field keywords>, subject to the exceptions noted on the statement types above:

Opcode	WITH Field	Operation
|+|		true binary add; byte <a> is added to byte <b>.

|+| CONE true binary add; a 1 is added to the sum of bytes <a> and <b>.

|+| COUT true binary add with carry out into S3.

|-| two's complement add; byte <a> is added to the complement of byte <b>.

|-| NCIN one's complement add; byte <a> is added to the 1's complement of byte <b>.

|-| COUT two's complement add; byte <a> is added to the complement of byte <b>, and the carry-out is placed in S3.

|B| binary add; if S0 is 0, a true add is performed on bytes <a> and <b>, and if S0 is 1, a complement add is performed.

|D| decimal add; if S0 is 0, a true add is performed on bytes <a> and <b>, and if S0 is 1, a complement add is performed.

|X| exclusive OR.

|O| Inclusive OR.

|A| AND.

|N| AND NOT.

|C| exclusive OR with parity; byte <a> is exclusive or'ed to byte <b> and S4 is set to 1 if a parity error is detected on byte <a>.

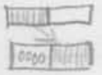
III.1.3.3 Gating Controls

Byte <a> can be gated by subscripting it with any of the following special identifiers:

1. L a byte is formed with bits 0-3 set to 0's and bits 4-7 from bits 4-7 of <a>.
2. H a byte is formed with bits 0-3 of <a> and bits 4-7 set to 0's.

← H

3. XH a byte is formed with bits 0-3 set to bits 4-7 of ⟨a⟩ and bits 4-7 set to 0's.



4. XL a byte is formed with bits 0-3 set to 0's and bits 4-7 set to bits 0-3 of ⟨a⟩

5. X a byte is formed with bits 0-3 set to bits 4-7 of ⟨a⟩, and bits 4-7 set to bits 0-3 of ⟨a⟩

Gating controls L and H may be used in conjunction with either ⟨a⟩ or ⟨b⟩. If they are used with ⟨b⟩, the only operations permitted are |+| or |X|. Gating controls XL,XH, and X may only be used in conjunction with ⟨a⟩ and only in conjunction with the operations |+| or |X|.

III.1.3.4 Status Setting or Resetting

Certain bits in the S-Register can be set or reset in the course of a byte arithmetic operation using one of the following with-field keywords:

S
123 45

1. S12 S1 is set to the value of the carry-out from bit 1 of the result; S2 is set to 1 if the result is not 0; otherwise it is unchanged.

2. S45 S4 is set to 1 if bits 0-3 of the result are 0; otherwise S4 is set to 0. If bits 4-7 of the result are 0, then S5 is set to 1; otherwise S5 is set to 0.

3. Z6 S4 is set to 1 if bits 0-5 of the result are 0; otherwise S4 is set to 0. S5 is set to 1 if bits 4-7 of the result are 0; otherwise S5 is set to 0.

III.1.3.5 Branching

Branching may be specified in the same manner as described for full-word arithmetic in section III.1.2.5.

III.1.3.6 Indirect Byte Operations

The indirect byte operations operate as follows:

1. Bytes in a Local or External storage word may be selected

using bits 4-5 of the T-Register for the <a> source and bits 6-7 of the T-Register for the <b> source.

2. Bits T4-5 and/or bits T6-7 may be incremented or decremented by 1
3. The execution of an instruction may be repeated until some branch condition is met. These branch conditions are determined by values of T4-5, T6-7, or S-Register bits.

Items 2 and 3 are only permitted in conjunction with indirect byte operations.

To specify indirect byte addressing, the reference to a word must be subscripted with (I). For example, if ALPHA and BETA are declared WORD then

Indirect addressing

```
ALPHA(I) <- ALPHA(I) |A| BETA(I);
```

will access the byte of ALPHA specified by bits 4-5 of the T-Register and AND this byte to the byte in BETA specified by bits 6-7 of the T-Register, replacing the byte obtained from ALPHA.

By specifying one of the following <with-field keywords>, the values of T4-5 and/or T6-7 can be modified, so that further execution of the same instruction will access different bytes in ALPHA and/or BETA:

1. INCA to increase T4-5
2. INCB to increase T6-7
3. DECA to decrease T4-5
4. DECB to decrease T6-7

For example,

```
ALPHA(I) <- ALPHA(I) |X| BETA(I) WITH INCA,DECB;
```

If T4-5 were initially 00 and T6-7 were 11, then byte 0 of ALPHA would be exclusive or'ed with byte 3 of BETA, and T4-5 would be increased to 01 while T6-7 would be decreased to 10. At the same time T0-3 would be set by the value of T4-5, since the target of the operation is the same as byte 0 of ALPHA. The setting of T0-3 from either T4-5 or T6-7, depending on the target is as follows:

T45(or T67)	T0-3
00	1xxx
01	x1xx
10	xx1x
11	xxx1

The x's signify no change to the existing bits.

III.1.3.7 Branching in Indirect Byte Operations

If indirect byte addressing is specified along with INCA, DECA, INCB, or DECB, then, and only then is indirect branching in effect.

Indirect branching operates as follows:

1. If no branch condition is met, the instruction address register remains unchanged, and the instruction is repeated.
2. If any one of the branch conditions is met, a branch occurs.

The following special identifiers may be used in the <bx> subscript field of indirect byte branching operations:

A0, A1, AB, 1B, 0B

The significance of these special identifiers is described in the System/370 Model 145 Functional Specifications (B).

An example of this is:

```
ALPHA(I) <- ALPHA(I) |X| BETA(I) WITH S12,INCA
            & GO TO NEXT(A0,S2,S3);
```

The branch condition is met only if one or more of the following are satisfied:

```
T45 = 11
S2  = 1
S3  = 1
```

Assume that the arithmetic instruction is executed once and that S2 is set to 1 as a result of the operation when S12 was specified in the with-field; also T45 is incremented to 01 and S3 remains equal to 0. The instruction is then re-executed because none of the branch tests that were made before S2 was set, were met. Since S2 is 1 this second time, there will be a branch to NEXT(0,1,0).

III.1.4 Word-Move Instruction

This instruction provides:

1. Moving a word, selected bytes in a word, or other special bytes from one register to another.
2. Displaying a word (via the A-Register) while performing a STOP operation.

It is used to address directly either one of its operands without using the P-Register. Thus, it is possible to obtain or store data in a location not covered by the current setting of the P-Register. One of the following with-field keywords may be used to designate whether the source or destination is to be addressed directly:

```
ALPHA <- BETA WITH DS;
BETA <- ALPHA WITH DD;
```

In these examples ALPHA is addressable with the current P-Register, and BETA is addressed directly. The DS indicates 'Direct Source', addressing the source directly without using the P-Register. The DD in the latter example indicates 'Direct Destination' addressing. One restriction in this second case is that the source may not be an External storage register, except the SPTL register.

III.1.4.2 Byte Selection Function

Instead of moving an entire word from one register to another, it is possible to move only 1, 2, or 3 bytes. To do this, a hexadecimal digit is appended to the DS or DD keyword, e.g., DS5 or DDF, to specify which bytes are to be selected, as indicated in the table below.

DD & DS

h	Bit Representation	Bytes to be moved
0	0000	none
8	1000	byte 0
4	0100	byte 1
2	0010	byte 2
1	0001	byte 3

DS4 means byte 1

DD7 means bytes 1,2,3

DD3 means bytes 2+3 ≡(H1)

0	1	2	3
1000	0100	0010	0001

Any combination is valid, e.g., 1010.

1100 ≡ "C" means 0,1 ≡(H0)

The keyword DSF indicates that all the bytes from the directly-addressed source are to be moved to the destination; this is equivalent to using DS. Bytes in the destination register not specified to receive a byte from the source are unaffected by this instruction.

III.1.4.3 STOP Function

The keyword STOP may be placed in the with-field of the word-move instruction.

ALPHA <- BETA WITH DS,STOP;

When the instruction is executed, the word in the source field is placed in the A-Register, and the normal word-move is performed. The instruction remains in the instruction register and is executed every CPU cycle. If the START button (on the CPU console) is pressed or an interrupt latch is set, the next address is specified by the branch,

& GO TO <label>(<bx>,<bh>,<bl>)

from which location the next instruction will be taken.

III.1.4.4 Branching

A limited branching capability is associated with the word-move instruction and is specified by writing:

& GO TO <label>(<bx>,<bh>,<bl>)

Permissible values for <bx>, <bh>, and <bl> are:

```
   <bx>
    0-3

   <bh>
    0,1,S0,S1,S2,S4,S6

   <bl>
    0,1,S3,S5,S7,Z0
```

These are described under section III.1.1.1 on the Branch Instruction.

III.1.5 Storage Reference Instruction

The storage reference instructions provide the ability to read or store data into Main or Control Storage. The general form of the instruction is:

```
   <destination> <- <source> WITH <with-field keywords>
                 & GO TO <label>(<bx>,<bh>,<bl>);
```

in which either the <destination> or the <source> is an identifier with a MAIN or CONTROL attribute having an implicit pointer given in the DECLARE statement, or an explicit pointer appearing in the instruction. The form of the <source> or <destination> with an explicit pointer is:

```
   <pointer> -> <identifier with MAIN or CONTROL attribute>
```

If the <destination> refers to storage, then the instruction is a store instruction; otherwise, if the <source> refers to storage, then the instruction is a read instruction. In the example,

```
                    ALPHA -> PROG <- BETA;
```

BETA is placed in PROG, to which ALPHA points. This will result in a store into MAIN or CONTROL storage depending on the attribute associated with PROG in the DECLARE statement. In the next example,

```
                    BETA <- ALPHA -> PROG;
```

the reverse operation is performed, namely the contents of PROG, to which ALPHA points, are placed in BETA. If the declaration of

PROG had been,

```
DECLARE PROG WORD MAIN BASED(ALPHA);
```

The statement could then be written:

```
BETA <- PROG;
```

A byte, half-word, or full-word can be accessed in MAIN or CONTROL storage in any of the following ways:

1. The attribute BYTE, HALF, or WORD may be included in the declaration of the identifier, such as PROG, above, e.g.,

   ```
   DECLARE PROG MAIN BASED(ALPHA) HALF;
   ```

2. The identifier may be written with one of the following special identifiers as a subscript:

PROG(B)	read or store a byte
PROG(H)	read or store a halfword
PROG(W)	read or store a full word

The second option will override any size attribute specified in the declaration of the identifier.

III.1.5.2 Store Operation

The data may be in either Local or External storage. In byte-storage operations the byte to be stored is byte 3 of the data register; in half-word storage operations it is bytes 2 and 3 of the data register; finally, in full-word operations the contents of the entire data register are used.

However, if TH, DECTH, or INCTH is specified in the with-field keyword list, then the bytes actually stored will depend on the setting of T0-3. These four bits provide a mask, such that the byte will be stored if the corresponding bit is set to 1.

T bits	Value	Byte to be Stored
0	1	0
1	1	1
2	1	2
3	1	3

Any combination of bytes can be stored. For example, if T0-3 are set to 1001, then bytes 0 and 3 from the data register are stored into bytes 0 and 3 of the addressed word in memory. Note: T0-3 are set to 0's after the bytes have been stored.

The use of either the DECTH or INCTH keywords specifies masking, as well as incrementing or decrementing as indicated in section III.1.5.4.

III.1.5.3 Read Operation

Data may be read from MAIN or CONTROL storage and placed in Local or External storage. In byte-read operations the byte is always placed in byte 3 of the <destination> word. In half-word read operations, the data is entered in bytes 2 and 3 of the <destination> register. In full-word read operations, the entire <destination> register is used. It is not possible to use the T-Register to select byte combinations for read operations as it was for store operations.

Designating TA or TB as with-field keywords in a read operation has the following effect:

> TA causes T4-5 to be set to the value of the two low-order bits of the storage address before that address is updated in the address register.
>
> TB causes the same action for T6-7.

In either case, even though byte selection can not be specified in conjunction with the read operation, T0-3 are set to 0.

III.1.5.4 Address and Count Update

A Local storage location may be used to address either MAIN or CONTROL storage; an even-odd pair of Local storage locations may be used to address MAIN or CONTROL storage and count the number of references at the same time. It is possible to increment or decrement the address register and to decrement the count register.

Updating such registers may be specified with the following with-field keywords:

INC	increment address register
DEC	decrement address register
DCNT	decrement count register
DECTH	decrement address register using the T-Register
INCTH	increment address register using the T-Register

For INC or DEC the address register is changed according to the type of read or store operation performed. When DCNT is specified, the count register is decremented by the same amount as was used for the address register. It is impossible to increment the count register.

Operation	INC	DEC	DCNT
byte	+1	-1	-1
halfword	+2	-2	-2
word	+4	-4	-4

When DECTH or INCTH are specified, the amount of the increment or decrement is determined by T0-3:

T0-3	Hex	Value of INCTH/DECTH/DCNT
0000	0	0
0001	1	1
0010	2	1
0100	4	1
0101	5	1
1000	8	1
1010	A	1
0011	3	2
0110	6	2
1100	C	2
0111	7	3
1110	E	3
1001	9	4
1011	B	4
1101	D	4
1111	F	4

For example,

```
INDATA <- PTR -> MAINPROG(W) WITH TA,INC,DCNT;
```

This is a word-read operation setting T4-5 according to the last two bits in the address register, incrementing the address register by 4, and decrementing the count register by 4. T0-3 is then set to 0.

III.1.5.5 Status Setting

Status bits in the S-Register can be set or reset according to the result obtained from updating the count register. By specifiying keywords S2, S45, or Z6 in the with-field,

S2: S2 is set to 1 if the count register is not 0; otherwise it is reset to 0.

S45: S4 is set to 1 if bits 0-3 of byte 3 of the count register are 0; otherwise S4 is set to 0. S5 is set to 1 if bits 4-7 of byte 3 of the count register are 0; otherwise S5 is set to 0.

Z6: S4 is set to 1 if bits 0-5 of byte 3 of the count register are 0; otherwise S4 is set to 0. S5 is set or reset the same way as was specified for S45.

If S2, S45, or Z6 is specified and the count register is not updated, then the S-Register will be set or reset according to the contents of the Z-Register. Note: The S-Register may not contain the value expected after this operation.

III.1.5.6 Update Only Instruction

```
    UPDATE <address register> WITH <with-field keywords>
                 & GO TO <label>(<bx>,<bh>,<bl>);
```

It is possible to update the address register and/or the count register without accessing storage. To do this the assignment portion of the storage reference instruction is replaced by UPDATE <address register>, where <address register> is a Local storage word. Keywords in the with-field will then designate how the registers are to be adjusted.

7

Update Address Register

INC1,INC2,INC4,INCTH
DEC1,DEC2,DEC4,DECTH

And Update Count Register

DCNT

Update Count Register Only

DCNT1,DCNT2,DCNT4,DCNTTH

The specifications S2, S45, or Z6 can also be used in conjunction with updating the count register.

III.1.5.7 Direct Storage Addressing

Instead of using an address register to access a location in MAIN or CONTROL storage, it is possible to address certain areas of this storage directly.

The locations in main storage which may be accessed directly are the low-order 256 bytes where the System/370 PSW, timer, CSW, etc. are located. To do this the special identifier, MMOD, is written:

MMOD(<constant>,<w-h-b>)

where <constant> designates the absolute address between 0 and 255, and <w-h-b> is either W, H, or B, a special identifier designating a word, half-word, or byte, respectively.

Three methods exist to access CONTROL storage:

1. CMOD(<constant>,<w-h-b>) refers to the module in which the instruction being executed is located. The address is the start of the module plus <constant>, a number between 0 and 255 inclusive. As above, <w-h-b> indicates one of the special identifiers, W, H, or B, specifying a full-word, half-word, or byte.

2. FMOD(<constant>,<w-h-b>) refers to the module of control storage whose starting address is hexadecimal FF00. The address is FF00 plus the value of <constant>, a number between 0 and 255 inclusive. <w-h-b> designate whether a

full-word, half-word, or byte is to be accessed.

3. <pointer> -> FMOD(<constant>,<w-h-b>) refers to the Fh module, where h is a hexadecimal constant. The address is Fh00 plus the value of the low-order byte of <pointer>. The value of h is determined from <constant>, whose value is between 0 and F, inclusive. <w-h-b> is used to specify the accessing of a full-word, half-word, or byte.

III.1.5.8 Branching

The following limited branching capability is provided in conjunction with storage reference instructions:

```
<bx>
0,1,2,3

<bh>
X,0,1,S0,S1,S2,S4,S6,M6

<bl>
X,0,1,S3,S5,S7,Z0,M7
```

Special identifiers, M6 and M7, refer to the low-order two bits of the updated storage address. The remaining special identifiers are described in section III.1.1.5.

III.1.6 Call Statement

The general form of the Call Statement is:

```
CALL <procedure name>.<label>(<bx>,<bh>,<bl>)
            USING <link register>
        GROUP (<keyword>,<keyword>);
```

When the Call statement is executed, the contents of the S-Register, the P-Register, and the address of the Call statement are placed in <link register>, a Local or External storage location, in the following order:

Register	<link register> bytes
S	0
P	1
Address	2-3

The contents of <link register> may be used by a subsequent Return statement to link back to the call statement.

The point to which the Call statement branches is the leg of the branch-set specified by,

```
        <procedure name>.<label>(<bx>,<bh>,<bl>)
```

If <procedure name> is the same as <procedure name> associated with the Call statement, then it may be ommited. The following are permissible settings for the subscript fields:

<bx>
0,1,2,3

<bh>
X,0,1,S0,S1,S2,S4,S6

<bl>
X,0,1,S3,S5,S7,Z0

The GROUP clause is used to specify how the P-Register is to be set at the new location. The parameters are described under the GROUP statement in Section III.3.4. This function is optional and may not be used in an instruction in which there is a branch to a different module.

III.1.7 RETURN Statement

The general form if the Return statement is:

```
      RETURN USING <link register>(<bx>,<bh>,<bl>);
```

Execution of the Return statement places information from the <link register> into the S-Register, P-Register, and address register.

Data created by a LINK declaration statement may also be used by a RETURN statement, see section III.2.3.

Permissible settings for the <bx>, <bh>, and <bl> fields are:

```
    <bx>
    0-15

    <bh>
    X,0,1,S0,S1,S2,S4,S6

    <bl>
    X,0,1,S3,S5,S7,Z0,I0,I1
```

A complete description of I0 and I1 can be found in the System/370 Model 145 Functional Specifications (B).

III.2 Declaration Statements

III.2.1 Register Declarations

The Register declaration is used to establish a symbolic name for a Local or External storage location. In this way it is possible to define words, half-words, bytes, and bits of such locations.

The SYN attribute is used for this. The name appearing in parenthesis following 'SYN' may be one of the following:

```
    LShh               (<bit>)
    EShh               (<byte>:<bit>)
    <identifier>       (<byte>)
```

LShh designates a Local storage location with hexadecimal address hh. The values of hh may be between 00 and FF inclusive and must always be written as two hexadecimal digits.

EShh designates an External storage register with address hh in hexadecimal.

<identifier> designates the symbolic name that has been declared previously for a Local or External storage location.

(<byte>), (<byte>:<bit>), or (<bit>) may be used to name a specific bit and/or byte. Permissible values for <byte> may be 0 to 3, and permissible values for <bit> may be 0 to 7.

For example,

example 1: DCL ALPHA SYN (ES03) WORD;

example 2: DCL BETA SYN (LS3F(3:7)) BIT;

In example 1 ALPHA is declared to be the third word in External storage.

In the second example, BETA is defined to be bit 31 of Local storage location 3F. The first subscript of LS3F, 3, designates the low-order byte of LS3F, while the subscript, 7, designates the last bit of byte 3.

```
+--------+--------+--------+--------+
| byte 0 | byte 1 | byte 2 | byte 3 |
+--------+--------+--------+--------+
3F
                           +-+-+-+-+-+-+-+-+
                           |0|1|2|3|4|5|6|7|
                           +-+-+-+-+-+-+-+-+
                                         BETA
```

Note: BIT, BYTE, or WORD may be written as part of the declaration; this must then coincide with the type of the SYN'ed variable.

III.2.1.2 Declaration of Structures

A structure may be used to give symbolic names to many parts of contiguous storage.

For example,

```
DCL  1 FRUIT WORD SYN(LS10),
       2 APPLES BYTE,
       2 PEARS  BYTE,
       2 ORANGES BYTE,
         3 FLORIDA BIT SYN(LS10(2:1)),
         3 NEWMEX BIT,
       2 PEACHES;
```

FRUIT is defined to be all of Local storage word 16, while portions of this word may be referenced as follows:

```
  FRUIT
+---------+---------+---------+---------+
| APPLES  |  PEARS  | ORANGES | PEACHES |
|  byte 0 | byte 1  | byte 2  | byte 3  |
+---------+---------+---------+---------+
  10

                 +-+-+-+-+-+-+-+-+
                 |0|1|2|3|4|5|6|7|
                 +-+-+-+-+-+-+-+-+
                  F N
                  L M
                  A
```

The locations of the substructures are assigned as they appear, but they may be located explicitly by using a 'SYN' clause; further assigning will continue from the new point with correct boundaries observed.

III.2.1.3 Multiple Declarations

If several names are to be given the same set of attributes, the declaration may be written:

```
          DECLARE (A,BCD,E,F) BIT SYN (LS01(0:0));
```

Thus A, BCD, E, and F all refer to the same bit in word 01 of Local storage.

III.2.2 MAIN and CONTROL Attributes

Names may be used to identify areas in MAIN or CONTROL storage in conjunction with word move instructions described in section III.1.4. When such an instruction is used, the MAIN or CONTROL storage name must have a pointer to the address in storage from which data is to be accessed. This pointer must be a Local storage location and may be implicitly associated with the name by the use of the BASED attribute.

```
          DECLARE  ALPHA WORD MAIN BASED(PTR);
```

Here, ALPHA is a name for MAIN storage and will be referenced as a word. Whenever ALPHA is used in the program, the pointer, PTR, previously declared in Local storage, will contain the proper address. The implicit pointer, PTR, may be overridden with an explicit pointer in the program, and the reference to the whole word may be overridden by subscripting ALPHA with (B) or (H), as indicated in section III.1.5.

III.2.3 The LINK Attribute

```
DCL  <identifier> LINK (<label> (0,0,0), <constant>);
                                to 1 1
                                 F X X
```

A location containing the <label> as the last two bytes and the <constant> as the first two bytes is formed with <identifier> as its name.

This word may be loaded into a location in Local storage and used in a Return statement to provide settings for the S-Register, P-Register, and the return address.

III.2.4 The EQU Attribute

The equivalence attribute is used to create a symbolic constant.

```
DECLARE <identifier> EQU ( <constant> );
```

The <identifier> may be used in place of a constant. Its value can only be changed by another declaration. The <identifier> does not correspond to a storage location; its purpose is to provide mnemonic capabilities for constants used as masks, offsets, etc.

III.2.5 The OFFSET Attribute

The OFFSET attribute is used to create a set of constant offset values for the elements in a structure. For example,

```
   DCL  1  TAB  OFFSET,
           2  AB  WORD OFFSET,
           2  XY  BYTE OFFSET,
           2  (BCD,MLQ)  HALF OFFSET,
           2  LASTBUNCH  WORD OFFSET,
              3  (L1,L2,L3,L4)  BYTE OFFSET;
```

The OFFSET attribute must appear at the first level of the structure. Elements in the structure shown above have a constant attribute with the following values:

```
    TAB = 0            LASTBUNCH = 9
    AB  = 0            L1 = 9
    XY  = 4            L2 = 10
    BCD = 5            L3 = 11
    MLQ = 7            L4 = 12
```

III.3 Special Control Statements

III.3.1 Constant Statement

```
        WORD   AT(<address constant>):<label>:<data>;
                                 or,
       TABLE  AT(<address constant>):<label>:
           <data>,<data>,...,<data>;
       ENDTABLE;
```

The WORD AT form may be used for <data> which occupy a single word; otherwise, the TABLE AT form is necessary. ENDTABLE is only to be used in conjunction with the TABLE AT form.

The <address constant> can have three forms:

1. An absolute value written in hexadecimal, decimal, or binary.
2. A hexadecimal value specifying the module in which to place the data, but not the specific location within that module is designated by:

   ('hhXX'X)

   where hh are hexadecimal digits from 00 to FF used to specify the module.

3. A hexadecimal value specifying the location in some module where the data are to be placed, without specifying the particular module. The form for this is:
   ('XXhh'X)
   where hh specifies the hexadecimal location.

The <data> can be either an instruction or a constant. A numeric constant is placed in a single word. A string constant is translated into its EBCDIC form, left-justified with blank fill, and placed in as many words as necessary to handle the entire string.

For example,

```
 TABLE AT ('XXE0'X): ALPHA:
     8,'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
 ENDTABLE;
```

translates to 8 words of memory starting at address E0 in an unspecified module and is labeled ALPHA. The following is the resulting storage assignment:

Address	Contents
XXE0	00000008
XXE4	C1C2C3C4
XXE8	C5C6C7C8
XXEC	C9D1D2D3
XXF0	D4D5D6D7
XXF4	D8D9E2E3
XXF8	E4E5E6E7
XXFC	E8E94040

III.3.2 Module Equate Statement

This statement is used to assign microprogram words or hexadecimal data to the same module, i.e., group of 64 control words. Its form is:

EQUATE MODULE <label1>,<label2>,...,<labelN>;

where <label1> must be in the current procedure. The labels may not have associated <bx>, <bh>, or <bl> fields, and must be qualified with a procedure name if they are located within another procedure.

For example,

```
EQUATE MODULE FIRST,PROG1.LEG1,PROG2.LEG1;
```

places the microprogram instructions with the labels FIRST, LEG1 in PROG1 procedure, and LEG1 in PROG2 procedure all in the same module. If any of these labels are part of a branch-set, then the entire branch-set will be placed in the same module.

III.3.3 The RESERVE Statement

This statement reserves locations in control storage, so that instructions or data will not be assigned to these locations, except with a WORD AT or TABLE AT statement.

```
            RESERVE <constant>;
                    or,
    RESERVE <constant> THRU <constant>;
```

where <constant> denotes a word address in control storage, i.e., it is a multiple of 4.

For example,

```
RESERVE 'E000'X THRU 'EFFC'X;
```

reserves all the words in modules E0 through EF. Data will not be assigned to these locations unless an explicit instruction is given to do so.

III.3.4 The GROUP Statement

```
GROUP (<keyword>,<keyword>);
        GROUP (<keyword>);
```

This statement is used to set the P-Register for addressing various locations in Local or External storage.

The three keywords associated with this statement are:

1. DLSd Direct Local Storage; where d is a digit between 0 and 7.

2. EXTd External Storage; where d is a digit between 0 and 7.

3. ILSd Indirect Local Storage; where d is a digit between 0 and 3.

The storage locations specified by the digit d are:

DLS or EXT

d	Addressable Storage (hexadecimal)
0	00-07
1	08-0F
2	10-17
3	18-1F
4	20-27
5	28-2F
6	30-37
7	38-3F

ILS (Local Storage)

d	Addressable Storage (hexadecimal)
0	00-0F
1	10-1F
2	20-2F
3	30-3F

EXT and ILS both set the high-order 4 bits of the P-Register, and therefore cannot be used in the same instruction. The difference between them is that EXT sets bit 3 of the P-Register (P3) to 1, while ILS sets P3 to 0. Note: If there is to be any indirect byte addressing, P3 must be 0.

DLS sets the low-order 4 bits of the P-Register. P3 must equal 0, so EXT may not be used.

Bibliography

A. WIRTH, Niklaus A Programming Language for the 360 Computers. Technical Report No CS 53, Computer Science Department, Stanford University, (June 1967).

B. IBM Corporation, SYS/370 Model 145 System Specifications, Section II, IBM Corporation, (November 1970).