```
+---------------+
! d i g i t a l !   I N T E R O F F I C E   M E M O R A N D U M
+---------------+
```

TO:	Mimi Chen	MR1-2/E37
	Ted Hess	MR1-2/E85
	Don Lewine	MR1-2/E85
	Patty Hardy	MR1-2/E37
	Ron McClean	MR1-2/E85
	Dave Nixon	MR1-2/E37
	Dave Wright	MR1-2/E37

DATE:	Oct 7, 1980
FROM:	Sara Murphy
DEPT:	Technical Languages
LOC:	MR1-2/E37
EXT:	231-5181
FILE:	INSTR1.RNO

SUBJ: CIS Instructions

The attached document is an update of the CIS specification. The only change is that all of the instructions are now specified to be EXTEND instructions. This change was made to improve performance on JUPITER.

Many of the instructions in this specification could only be used by COBOL if 9-bit ASCII were supported. There is currently no plan for COBOL to support 9-bit ASCII.

Some of the instructions in this specification could be used by COBOL to improve packed decimal and EBCDIC performance. Those instructions are:

MOVP
ADDP
CHOP
CMPP
CVTPB
CVTBP
SUBP
CVTEP
CVTPE
MOVC
CMPC
ASHP

Please note that appendix B of the specification specifies extensions to be made to the BIS instruction set. These extensions can be utilized by COBOL regardless of whether 9-bit ASCII is supported.

** COMPANY CONFIDENTIAL **   DO NOT DUPLICATE

```
+---------------+
! d i g i t a l !   I N T E R O F F I C E   M E M O R A N D U M
+---------------+
```

TO: List

```
                                  DATE:  7-Oct-80
                                  FROM:  SARA MURPHY X5181
                                         BILL KOHLBRENNER X1100
                                  DEPT: Large System
                                          Software Engineering
                                  LOC:   MR1-2/E37
                                  FILE:  INSTR.RNO
```

SUBJ: NEW -10/20 COMMERCIAL INSTRUCTIONS
REVISION: 12

This document describes a set of commercial instructions for the DECSystem-10/20. This specification was approved by the 10/20 Architecture Committee on March 22, 1979.

Table of Contents

1.0 MOTIVATION AND GOALS

The commercial instructions being proposed are designed to make COBOL object code run fast. This goal was of primary importance in defining the instructions. The COBOL performance required can not be achieved by simply speeding up the -10 BIS because of the setup overhead and generality of those instructions.

Other goals that were taken into consideration were consistency with the rest of the KL10-model B architecture and applicability to other languages.

Our primary interest was in the potential performance of these instructions on new machines. It was also important that the instructions improve COBOL performance on KL-10 model B machines.

2.0 OVERVIEW

These new instructions are tied to a general software shift to 9-bit bytes, with inclusion of a packed decimal (2 digits/byte) data type.

The instructions being proposed fall into two classes:

1. Character string manipulation instructions, which manipulate strings of 9-bit bytes.
2. Decimal arithmetic instructions (both for numbers represented in 9-bit display format and for numbers represented in packed decimal format)

In general the instructions being proposed take two operands and perform memory to memory operations.

A number of addressing modes are available for addressing the operands of these instructions. These addressing modes are described in detail below. Instructions may be 2,3, or 4 words long depending on the addressing modes used for their operands.

The following instructions are being proposed. They are described in detail in sections 6 and 7 below.

Character String Manipulation Instructions

MOVC	-	Move Characters
CMPC	-	Compare Characters
MOVCV	-	Move Characters Variable length
CMPCV	-	Compare Characters Variable length

Numeric Display (Trailing overpunch) Instructions

CMPND	- Compare Numeric Display
ADDND	- Add Numeric Display
SUBND	- Subtract Numeric Display
MOVND	- Move Numeric Display
ASHND	- Arithmetic Shift Numeric Display
CVTNDB	- Convert Numeric Display to Binary
CVTBND	- Convert Binary to Numeric Display
TLGLND	- Test for Legal Numeric Display

Packed Decimal Instructions

CMPP	- Compare Packed
ADDP	- Add Packed
SUBP	- Subtract Packed
MOVP	- Move Packed
ASHP	- Arithmetic Shift Packed
CHOP	- Clear high order packed
CVTPB	- Convert Packed to Binary
CVTBP	- Convert Binary to Packed
CVTNDP	- Convert Numeric Display to Packed
CVTPND	- Convert Packed to Numeric Display
CVTEP	- Convert EBCDIC numeric display to Packed
CVTPE	- Convert Packed to EBCDIC numeric display
TLGLP	- Test for legal packed decimal

The instructions listed above are referred to as CIS ("Commercial Instruction Set") instructions.

These complement some of the less time-critical BIS instructions that will still be used, such as EDIT and MOVST (which does character set conversion). A number of extensions will be made to the BIS instructions. These extensions are described in appendix B.

3.0 EFFECTIVE ADDRESS CALCULATION FOR 9-BIT BYTES

3.1 Goals For The Address Calculation Scheme

The following goals were taken into consideration when deriving an addressing scheme for addressing 9-bit byte strings:

1. It must be possible to address all of virtual memory

2. Simple addressing operations within a single section should work without indexing or base registers or execution of extra instructions to perform address calculation

3. There should be a simple representation that can be used to describe the location of a parameter or a dynamically allocated string anywhere in the virtual address space.

4. Singly subscripted array references to character strings should run as fast as possible

5. References to elements in multi-dimensional character string arrays should be "straightforward"

6. The scheme used should be consistent with the effective addressing rules used by the existing instruction set. (This is important for performance reasons. DOLPHIN has hardware that calculates an effective address from the I,X, and Y fields of every instruction before the opcode is decoded. These instructions should take advantage of that calculation if possible.)

7. We should allow for future expansion.

3.2 Addressing Modes

The instructions address byte strings on their leftmost byte.

The address of a 9-bit byte is composed of the following:

1. The 30-bit virtual address of the word that contains the byte

2. The 2-bit byte offset within the word. The leftmost byte within a word is byte 0, the next is byte 1, etc

The values of these two components may be specified in a number of different ways which are described below.

An operand that refers to a 9-bit byte string may be either one or two words long. The first word of each operand has the following format:

```
0        8 9 10 1112 13 14        17 18                          35
--------------------------------------------------------------------
!  XXXXX ! M  ! O  !I !    X      !             Y                 !
--------------------------------------------------------------------
```

Bits 0-8 are never part of the operand. (The bits preceding the first operand of an instruction contain the opcode. The bits preceding the second operand are used as length fields. Their interpretation is different for different opcodes and is described under each individual instruction.)

A 30-bit word address is always calculated from I,X and Y in the same way as for all other PDP-10 instructions. We will refer to this address as E.

There are a number of addressing modes that are specified by the values in M and O.

3.2.1 Direct Mode -

This addressing mode is used for all operands whose addresses are known at compile time. For COBOL-79, this includes all scalars except for subroutine parameters.

The word address of the byte is the effective address calculated from I,X, and Y. The byte offset is equal to the value in the O field.

This mode is indicated by a 0 in the M field.

3.2.2 Deferred Mode -

This addressing mode is used for operands whose addresses are not known at compile time, such as parameters or strings in dynamically allocated storage.

The contents of the location addressed by I,X, and Y are interpreted as a 32-bit byte address. Bits 4-33 of this word specify a 30-bit global word address. Bits 34-35 of this word specify a byte offset. Bits 0-3 of this word must be zero. If they are not, an MUUO trap will occur.

This mode is indicated by a 1 in the M field and a 0 in the O field.

3.2.3 Direct Subscripted Mode -

This mode is used for addressing elements of 1-dimensional arrays whose base addresses are known at compile time.

The value of E and the contents of the O field specify the base address of an array. As in "direct mode", the word address of the base of the array is equal to E and its byte position is specified by O.

The next word in the instruction stream is also used in specifying the operand. It has the format:

```
 0 1             12 13 14    17 18                          35
-------------------------------------------------------------
!0! ELEM SIZE     !I2!    X2  !     Y2                        !
-------------------------------------------------------------
```

Bits 1-12 specify the size of an array element in bytes.

Let E2 be the word address calculated from I2, X2, and Y2. The word addressed by E2 contains a subscript. This subscript may be any signed 36 bit binary number.

The instructions assume that the base address specified by E and O is the address of element 1 of the array. They compute a byte offset into the array using the formula:

(Contents of(E2) - 1) * ELEM SIZE

The byte offset calculated is added to the byte address of the base of the array. This address calculation can cross section boundaries, regardless of whether the base address of the array was local or global.

Bit 0 of the second word is reserved and must be zero. If it is set the instruction is aborted and an MUUO trap occurs.

The compiler can handle multiple subscripts, items greater than 4095 bytes, etc by generating code that computes the byte offset and having E2 point to that computed offset. The ELEMENT SIZE field should then be 1.

The instructions do not perform any range checking on subscripts. If range checking of subscripts is desired, a compiler must insert code to do this range checking before the instruction that references an array element.

The subscript address may be either local or global. It is legal to refer to an AC (except AC 0) as a subscript, and that AC may be specified as either a local address 1 - 17, or a global address 1 - 17 in section 1. If the subscript address is AC 0, results are indeterminate.

This mode is indicated by a 2 in the M field.

3.2.4 Deferred Subscripted Mode -

This mode is used for addressing elements of arrays when the base address of the array is not known at compile time (eg arrays that are parameters or are allocated in dynamic storage).

In this case the word addressed by E contains a 32-bit byte address of the base of the array. This byte address has the same format as is used in "deferred mode".

As in direct subscripted mode, the next word in the instruction stream is used to specify the element size and the location of the subscript.

Note that the element size is a part of the instruction and therefore must be known at compile time. This is always true for COBOL programs. In languages where this is not true, the byte offset for an array element must be calculated before the operand is addressed and an element size field of 1 used.

This mode is indicated by a 1 in the M field and a 2 in the O field.

3.2.5 Immediate Mode -

"Immediate mode" has two different interpretations, depending on whether the instruction being executed is a character string manipulation instruction (MOVC, CMPC, MOVCV, CMPCV) or a decimal arithmetic instruction.

1. For character string manipulation instructions

   In this case, immediate operands are used to represent strings in which all characters are identical, for example a string of spaces.

   The operand is a string in which all bytes are identical to the 9-bit byte contained in bits 27 - 35 of the effective address E calculated from I, X, and Y.

2. For the decimal arithmetic instructions

   In this case, immediate operands are used to represent small integer constants.

   There are two types of decimal arithmetic instructions: 9-bit display decimal and packed decimal (see section 7). For both type of instructions, an immediate operand consists of the digits contained in bits 18-35 of E with zero fill supplied on the left.

For 9-bit display decimal instructions, bits 18-35 of E contain 2 decimal digits with an overpunched sign, supporting numbers in the range -99 to +99. For packed decimal instructions, they contain 3 decimal digits plus a sign nibble, supporting numbers in the range -999 to +999. See the sections below on decimal arithmetic.

The length field associated with a decimal immediate operand must always be 2 (this count is a byte count in the packed decimal instructions). If it is not 2, the results are machine-dependent.

There are a number of places where an immediate operand is illegal (e.g. as the destination of a move characters instruction). If an immediate operand is specified in one of these places, an MUUO trap will occur. The places where immediate operands are illegal are specified under the individual instructions.

This mode is indicated by a 1 in the M field and a 1 in the O field.

3.2.6 Summary Of Mode Bit Assignments -

The interpretation of the M and O fields for 9-bit operands is summarized in the following table:

M	O	Mode
0	Byte Position: 0,1,2, or 3	Direct Mode
1	0	Deferred Mode
1	1	Immediate Mode
1	2	Deferred Subscripted
1	3	Illegal. MUUO trap will occur.
2	Byte Position: 0,1,2, or 3	Direct Subscripted Mode
3	0,1,2, or 3	Illegal. MUUO trap will occur.

3.3 Examples

The following examples all use the MOVC ("Move Characters") instruction which is described in detail in section 5.1.1. In this instruction, bits 0-8 of the second operand contain the number of characters to be moved.

3.3.1 An Example Of "Direct Mode" Addressing -

The following is an example of an instruction that moves a character string between a source and destination whose addresses are known at compile/link time.

```
0        8 9 10 1112 13 14          17 18                          35
----------------------------------------------------------------------
! MOVC    ! 0  ! 1  !0 !    0        !          2000                 !
----------------------------------------------------------------------
!     7   ! 0  ! 3  !0 !    0        !          2500                 !
----------------------------------------------------------------------
```

This instruction moves a string of 7 9-bit bytes that starts in byte 1 (the second byte) of word 2000 to a destination that starts at byte 3 (the fourth byte) of word 2500. Both the source and destination addresses are in the same section as the instruction.

3.3.2 An Example Of Immediate Addressing -

The following is an example of an instruction that fills a character string with a character specified by an immediate operand.

```
0        8 9 10 1112 13 14          17 18             26 27        35
----------------------------------------------------------------------
! MOVC    ! 1  ! 1  !0 !    0        !                 !     040     !
----------------------------------------------------------------------
!     5   ! 0  ! 3  !0 !    0        !               2500            !
----------------------------------------------------------------------
```

This instruction fills a 5 byte string that starts at byte 3 of location 2500 with ASCII spaces (040).

3.3.3 An Example Of "Direct Subscripted Mode" Addressing -

The following is an example of an instruction that moves a string which is an element in a table whose base address is known at compile/link time. The destination to which this string is being moved is a scalar whose address is known at compile/link time.

Assume that the COBOL declaration for the table referenced is:

```
01 TABLE
  02 TABLE-ELEMENT OCCURS 20 TIMES
    03 TE-1 PIC X(2)
    03 TE-2 PIC X(7)
    03 RE-3 PIC X(17)
```

The following instruction executes a

```
     MOVE TE-2(I) TO D
```

assuming that the table begins at byte 0 of word 2000 and the destination D is at byte 3 of location 2500.

```
0        8 9 10 1112 13 14        17 18                         35
-------------------------------------------------------------------
!   MOVC ! 2  ! 2  !0 !   0       !          2000                !
-------------------------------------------------------------------
!0!      32        !0 !   0       !          6000                !
-------------------------------------------------------------------
!    7   ! 0  ! 3  !0 !   0       !          2500                !
-------------------------------------------------------------------
```

This instruction moves an element in a table in which each table element is 32 (octal) bytes long. Location 6000 contains the subcript for the element that is being moved. This subscript is a signed binary number. The first occurrence of the field that is being moved is at the third byte (byte 2) of word 2000.

The field being moved is 7 bytes long. The destination is at byte 3 (the fourth byte) of word 2500. Both the source and destination addresses are in the same section as the instruction.

3.3.4 An Example Of "Direct Mode" Addressing To Another Section -

The following is an example of an instruction that moves a charcter string between a source and a destination whose addresses are known at compile/link time, where the source and destination are both in sections other than the one containing the instruction.

```
0        8 9 10 1112 13 14        17 18                         35
-------------------------------------------------------------------
! MOVC   ! 0  ! 1  !1 !   0       !          1000                !
-------------------------------------------------------------------
!    7   ! 0  ! 3  !1 !   0       !          1001                !
-------------------------------------------------------------------
```

This instruction uses indirect pointers at locations 1000 and 1001 of the section containing the instruction. Assume that the contents of those locations is:

```
1000 :          3,,2000
1001 :          7,,2500
```

This instruction moves a string of 7 9-bit bytes that starts in byte 1 (the second byte) of word 2000 in section 3 to a destination that starts at byte 3 (the fourth byte) of word 2500 in section 7.

4.0 OPCODE ASSIGNMENTS

The CIS instructions will be executed by means of an EXTEND instruction, where the 2, 3, or 4 word CIS instruction is located at the effective address E0 of the EXTEND.

Opcode assignments are:

Instruction	Extend Opcode
MOVP	34
CMPP	35
ADDP	36
SUBP	37
CHOP	40
ASHP	41
TLGLP	42
CVTPB	43
CVTBP	44
CVTEP	45
CVTPE	46
MOVC	47
CMPC	50
MOVCV	51
CMPCV	52
MOVND	53
CMPND	54
ADDND	55
SUBND	56
ASHND	57
TLGLND	60
CVTNDB	61
CVTBND	62
CVTNDP	63
CVTPND	64

(Note that CMPSO and CMPST which are proposed in appendix B have EXTEND opcodes 32 and 33 respectively.)

5.0 GENERAL CONVENTIONS

5.1 Multi-word Instruction Format

The new instructions are 2,3, or 4 words long. All words of an instruction must be in the same section.

It is possible to perform an XCT of these instructions.

5.2 Interrupt Handling

All of the instructions use AC 0 to store state information during the handling of interrupts. Except where noted, the initial contents of AC 0 are ignored, and the final contents of AC 0 are 0.

5.3 MUUO's

There are a number of illegal forms of the CIS instructions that result in MUUO's. All of the conditions that result in MUUO's are detected during instruction setup, before data has been destroyed.

When an MUUO occurs while executing an CIS instruction, bits 15-17 of the PC word will indicate how to find the start of the instruction that failed. The address of the start of the instruction will be the value of PC minus the value in bits 15-17 minus 1.

If an MUUO occurs while executing an XCT or EXTEND of an CIS instruction, the PC will point to the instruction after the XCT or EXTEND and bits 15-17 of the PC word will contain 0.

5.4 Operands In AC's

In general the CIS instructions will work for operands that are in AC's other than AC 0. An operand in an AC can be specified by having the "word" part of the effective address refer to local addresses 1-17 or global addresses 1-17 in section 1.

If an operand refers to AC 0, the results are unspecified (they may depend on whether or not the instruction was interrupted, caused a page fail, etc).

5.5 Section 0

The CIS instructions will work in all sections, including section 0.

Note that in the subscripted direct and subscripted deferred addressing modes, if the base address of an array is in section 0, then subscripting will wrap around in section 0 rather than causing a reference to section 1.

5.6 Operations Crossing Section Boundaries

In the subscripted addressing modes, subscripts will always cross section boundaries, regardless of whether the base address of the array is local or global. The only exception to this is when the base address is in section 0.

Note that a large negative subscript to an array that starts in section 1 may cause a reference to section 0.

5.7 Overlapping Operands

All of these instructions, except ASHP and ASHND, operate on non-overlapping operands and on operands that are identical to each other (ie operands that have the same length and base address). The character string manipulation instructions also operate on overlapping operands in the case where the start of the destination operand precedes the start of the source operand. The instructions do not test for overlap, and if illegal overlap exists the result stored in the receiving field is unspecified.

ASHP and ASHND only operate on non-overlapping operands. If operands for ASHP or ASHND overlap or are identical to eachother, the results stored in the receiving field are unspecified.

The instructions will not work if there is overlap between the destination operand and any locations that are used in address calculations for the operands. In this case results are indeterminate.

5.8 Results Of Comparison Operations

Comparison instructions compare the first operand to the second operand. The result of the comparison is stored in bits 0, 1, and 2 of the flag word (Overflow, Carry 0, and Carry 1 flags). These bits of the flag word can be tested with JFCL. The comparison results are stored as follows:

bits 0,1,2	meaning
100	operand 1 less than operand 2
010	operand 1 greater than operand 2
001	operand 1 equal to operand 2

Most of the decimal arithmetic instructions compare the result value with 0 and store the result of that comparison (note that 0 is the second operand) in the same bits of the flag word.

5.9 Arithmetic Faults

A trap 3 will occur when a numeric display or packed decimal result does not fit in the destination field. When this trap occurs, high order digits will be truncated, the truncated result will be stored, and the PC will point to the instruction following the one that caused the overflow.

Bits 15-17 of the PC flags word will be used to specify the length of the instruction that overflowed. These bits will contain length minus one. (Three bits are used to allow for future definition of instructions that are more than 4 words long).

If a trap occurs when an XCT or EXTEND of an CIS instruction is performed, then the PC will point to the instruction after the XCT and bits 15-17 of the PC word will contain 0.

The packed decimal and numeric display instructions (other than compare or test instructions) will always set flags 0 - 3, and 11 and 12.

Flags 0 - 2 (OV, CRY0, CRY1) indicate the sign of the result (See the previous section)

If the result of a packed decimal or numeric display instruction fits in the destination, then flag bits 3, 11, and 12 (FOV, FXU, DCK) are set to 000. If it does not, then flag bits 3, 11, and 12 are set to 100 (ie FOV is 1, FXU and DCK are 0).

Note that the numeric display and packed decimal compare instructions set flags 0 - 2 to indicate the result of the comparison. They do not set flags 3, 11, and 12.

5.10 PXCT

PXCT of MOVC, CMPC, MOVCV, and CMPCV will work on future machines (not on the KL, since none of the instructions can be executed in the monitor on the KL).

The AC bit assignments for PXCT of these instructions will be consistent with those defined for the EXTEND instructions. Thus the following AC field values will be available:

AC Field	References in Previous Context
0001	Destination string
1001	Destination string; and all references made in doing the effective address calculation for the destination string
0010	Source string
1010	Source string; and all references made in doing the effective address calculation for the source string
0011	Source string and destination string
1011	Source string and destination string; and all references made in doing the effective address calculations for both source and destination

In the above table "all references made in doing the effective address calculation" includes the pointers used in deferred mode and deferred subscripted mode addressing, the subscripts used in direct subscripted mode and deferred subscripted mode addressing, and all indirect words and index registers referenced.

For MOVCV and CMPCV, the ACs that specify the string lengths will always be in the monitor address space.

A PXCT of an CIS instructions other than MOVC, MOVCV, CMPC, CMPCV will have unspecified results.

6.0 CHARACTER STRING MANIPULATION INSTRUCTIONS

Each of these instructions has two operands, each of which is a string of 9-bit bytes. The instructions may be used to manipulate any character set that is stored in 9-bit bytes.

A length of zero means that the operation will be performed on the null string. Note that side effects (such as AC 0 being set to zero) will occur when operations are performed on the null string.

6.1 Fixed Length Character String Manipulation Instructions

These instructions always deal with equal length strings. Bits 0-8 of the second operand of each of these instructions contain the number of bytes in the strings being operated on. A maximum string length of 511 bytes can be handled.

6.1.1 MOVC (Move Characters) -

This instruction moves a string of 9-bit characters.

Its first operand specifies the location of the character string to be moved and its second operand specifies the destination. Bits 0-8 of the second operand specify the number of characters to be moved.

If the first operand of this instruction is an immediate operand, then this instruction performs a "fill" operation. That is, all characters of the destination string are set to the character contained in bits 27-35 of the effective address computed for the first operand. (See the discussion of immediate operands in section 3) Note that a variable fill character may be specified by using indexing or indirection in the immediate operand.

It is illegal for the second operand of this instruction to be an immediate operand.

6.1.2 CMPC (Compare Characters) -

This instruction compares two equal length character strings. The result of the comparison is stored in the flag word (see section 5.8).

The two operands of this instruction specify the character strings to be compared. Either (or both) of these operands may be an immediate operand, representing a string in which all characters are identical to the character contained in bits 27-35 of the effective address computed for that operand.

Bits 0-8 of the second operand of this instruction specify the number of characters to be compared. If this length field is zero, then the result of the comparison is "equal".

The character strings are compared according to their binary values, with each byte treated as a 9-bit unsigned binary quantity.

The result of the comparison is stored in the flag word as described in section 5.8.

At completion, the contents of AC 0 is the number of initial characters that are equal. For example, if the the first characters of the strings differ, AC 0 contains 0; if the strings are identical AC 0 contains the value of the LENGTH field.

Note that this instruction can be used to scan over all initial occurrances of a particular character. If one of the operands of this instruction is an immediate operand, then, at completion, the contents of AC 0 is the number of initial characters in the other operand that are equal to the immediate character. For example, if the the first character of the other operand is not equal to the specified character, AC 0 contains 0; if the other operand is entirely composed of the specified character, AC 0 contains the value of the LENGTH field.

6.2 Variable Length Character String Manipulation Instructions

These instructions are identical to the character string manipulation instructions described above except that the lengths of the strings are specified in AC'S rather than in the instructions themselves. Also, these instructions can operate on strings of different lengths.

Bits 1-4 and bits 5-8 of the second operand of each of these instructions specify AC's which contain the lengths of each of the operands. Bits 4-35 of each of these AC's specify a string length between 0 (the null string) and 2**28 - 1 (the maximum number that can fit in the 27 bits in AC 0 that are available for storing state information when an interrupt occurs). A length outside of this range is an error condition which results in an MUUO.

When operations are performed on strings of different lengths, it is assumed that the shorter string is left justified and padded on the right. The pad character to be used is specified in the following way:

1. If Bit 0 of the second operand of the instruction is 0, then the pad character is ASCII blank (040). Both FORTRAN and COBOL specify that padding with blanks should be used.

2. If bit 0 of the second operand is 1, then the pad character is specified by bits 27-35 of AC 0. One use of this feature will be for EBCDIC blanks in COBOL programs.

These instructions will be useful in COBOL for programs containing reference modification (ie specification of substrings by means of subscripts). However this is a new feature in COBOL-79 and is not yet used. These instructions are essential for other programming languages, such as Fortran, APL, BASIC, and PL/1, in which strings are variable length. They also will be very useful for DBMS.

6.2.1 MOVCV (Move Characters Variable Length) -

This instruction moves a string of 9-bit characters.

Its first operand specifies the location of the character string to be moved and its second operand specifies the destination.

Bits 0-8 of the second operand of this instruction are composed of the following fields:

```
0 1     4 5     8
-----------------
!F! A1   !  A2   !
-----------------
```

A1 specifies an AC that contains the length of the character string to be moved. A2 specifies an AC that contains the length of the destination. A1 and A2 may refer to the same AC. The values in A1 and A2 are not changed by this instruction.

If the source string is longer than the destination, it will be truncated on the right. If the source string is shorter than the destination, it will be stored left justified and padded on the right. The pad character to be used is specified in the following way:

1. If the F bit in the second operand of the instruction is 0, then the pad character is ASCII blank (040).

2. If the F bit in the second operand is 1, then the pad character is specified by bits 27-35 of AC 0.

The first operand of this instruction can be an immediate operand. In this case the source string is composed of identical characters and has the length specified by A1. It is illegal for the second operand of this instruction to be an immediate operand.

(Note that a destination can be filled with a single character C in two different ways: (1) By having a source operand of length 0, with C specified as the pad character. (2) By using an immediate operand C as the source operand)

6.2.2 CMPCV (Compare Characters Variable Length) -

This instruction compares two character strings. The result of the comparison is stored in the flag word (see section 5.8).

The two operands of this instruction specify the strings to be compared.

Bits 0-8 of the second operand of this instruction are composed of the following fields:

```
0 1    4 5    8
----------------
!F! A1   !  A2  !
----------------
```

A1 specifies an AC that contains the length of the first character string and A2 specifies an AC that contains the length of the second character string. A1 and A2 may refer to the same AC. The values in A1 and A2 are not changed by this instruction.

If the lengths of the strings differ, the comparison is performed as if the shorter string were left justified and padded on the right. The pad character to be used is specified in the following way:

1. If the F bit in the second operand of the instruction is 0, then the pad character is ASCII blank (040).

2. If the F bit in the second operand is 1, then the pad character is specified by bits 27-35 of AC 0.

If one of the operands is the null string (ie has length 0), then it is treated as a string of all pad characters.

The character strings are compared according to their binary values, with each byte treated as a 9-bit unsigned binary quantity.

The result of the comparison is stored in the flag word as described in section 5.8.

At completion, the contents of AC 0 is the number of initial characters that are equal. For example, if the the first characters of the strings differ, AC 0 contains 0; if the strings are identical AC 0 contains their length; if the strings are identical except for trailing pad characters in the longer string, AC 0 contains the length of the longer string.

Either of the operands of this instruction may be an immediate operand. In that case, that operand is composed of identical characters and has the length specified by the corresponding AC.

7.0 DECIMAL ARITHMETIC INSTRUCTIONS

The decimal arithmetic instructions come in two groups. One group operates on numeric display data, consisting of 9-bit byte strings containing ASCII numeric bytes plus an encoded sign. The second group of instructions operates on packed decimal data consisting of 9-bit byte strings containing two four bit decimal digits (nibbles) per byte, with an algebraic sign encoded in the rightmost nibble.

Both groups of instructions have a common instruction format, and a similar set of opcodes.

Most of the decimal arithmetic instructions have two operands whose addresses are represented in the form described in section 3 above.

Those decimal arithmetic instructions that operate on two byte string operands include 2 length fields, each of which is 4 bits long. Each length field contains the length of the corresponding operand, in bytes. A length of zero is interpreted to mean 16 bytes. Thus they can operate on operands of different lengths, up to a maximum of 16 digits in the case of numeric display data, and 31 digits in the case of packed decimal data.

Both groups of instructions provide for arithmetic operations and data movement on signed numeric quantities. The instructions contain a sign control bit, S, which governs whether the result is stored with the normal algebraic sign encoding or whether the result is stored without a sign, ie, 'unsigned'. When a result is stored without a sign, it is, in effect, being stored as an absolute value; on a subsequent fetch, the operand is considered positive. The sign encodings are explained in following sections.

The sign control bit and the lengths of the operands are contained in bits 0-8 of the second operand of each instruction. The format of these bits is:

```
0 1    4 5    8
-----------------
!S! L1  ! L2    !
-----------------
```

L1 and L2 specify the lengths in bytes of the respective operands.

The S bit governs the storing of the sign encoding. (Zero in the S bits indicates "unsigned"; one indicates "signed".)

When lengths are different, operands are right justified with leading zeroes.

All of the instructions interpret minus 0 as identical to plus 0. CHOP is the only instruction that will ever generate -0 as a result.

Illegal numeric digits and illegal sign encodings are ignored and give machine dependent results. The only exception to this is the TLGLP and TLGLND instructions, which test for legal packed decimal and numeric display.

Comparison instructions store the result of the comparison in the flag word, bits 0 - 2 (OV, CRY0, CRY1).

The remaining decimal instructions compare the result operand, as stored, to 0 and store the result of that comparison in bits 0 - 2 of the flag word. An intermediate result, before it has been stored, is subject to possible high order truncation, which can leave the remainder equal to 0 and is also subject to having its sign changed from negative to positive by action of the S bit. Only after any truncation and sign change is the result compared to 0.

All of the packed decimal and numeric display instructions that store a result (i.e. all except compare or test instructions) set flags 3, 11, and 12 (FOV, FXU, DCK) to indicate whether the result fits in the destination word. These flags are set to 000 if the result fits, 100 if it does not.

When the result of a decimal arithmetic operation does not fit in the destination a trap 3 will occur. (see section 5.9 above).

Overflow in arithmetic operations leaves a properly truncated result. Truncation of high order significant digits during store operations stores the low order digits correctly.

7.1 Numeric Display Instructions

The numeric display instructions operate on decimal integers that are represented as strings of 8-bit ASCII characters, where each character is stored right justified in a 9-bit byte and the leftmost bit of each byte is 0. The addresses of numeric display operands are represented in the form described in section 3.

In a numeric display number, the sign is represented by a "trailing overpunch". This means that the last byte of the number contains both digit and sign information. The overpunch signs are:

ASCII	OCTAL	VALUE REPRESENTED
0 - 9	060 - 071	+0 TO +9
A - I	101 - 111	+1 TO +9 (default plus)
J - R	112 - 122	-1 TO -9 (default minus)
{	173	+0 (default +0)
[	133	+0
?	077	+0
}	175	-0 (default -0)
]	135	-0
:	072	-0
!	041	-0

The numeric display instructions accept numbers that use any of the overpunch characters listed above; however, the results generated by those instructions will always use A to I for +1 to +9, J to R for -1 to -9, and { and } for +0 and -0, when the S bit is set to 1. When the S bit is set to 0, ie, 'unsigned', the rightmost byte is stored as an ASCII 0 to 9 byte.

The instructions assume that each 9-bit byte (except for the last byte) in each operand contains a value in the range octal 060 to 071, representing the digits 0 to 9. The instructions ignore the contents of the leftmost 5 bits in each byte (other than the last) of their operands. They always set the leftmost 5 bits to 00011 (legal ASCII). If the right 4 bits are outside the range 0000 to 1001 the results are unspecified.

The arithmetic operations on numeric display data set bits in the flag word to indicate whether the result, as stored, is greater than, less than, or equal to zero.

The source operands for these instructions can usually be immediate operands (see section 3). In an immediate operand, bits 18 to 35 of the effective address are treated as two 8-bit ASCII characters stored in two 9-bit bytes in 'signed format', allowing an immediate value in the range -99 to +99. The length field for an immediate operand must always be 2.

7.1.1 CMPND (Compare Numeric Display) -

This instruction compares two numbers represented in numeric display form. The result of the comparison is stored in the flag word (see section 5.8). It treats plus and minus zero as identical.

This instruction has two numeric display operands that specify the numbers to be compared.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1   4 5    8
-----------------
!0!  L1  !  L2  !
-----------------
```

L1 is the number of digits in the first number; L2 is the number of digits in the second number.

Either of the operands of this instruction can be immediate operands.

7.1.2 ADDND (Add Numeric Display) -

This instruction adds two numbers represented in numeric display form. The result is left in the location of the second operand.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

This instruction has two numeric display operands. It adds that values of these two numbers and leaves them in the location of the second operand.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1   4 5    8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 is the number of digits in the first operand; L2 is the number of digits in the second operand.

S specifies the sign encoding of the result.

The first operand of this instruction can be an immediate operand. It is illegal for the second operand of this instruction to be an immediate operand.

7.1.3 SUBND (Subtract Numeric Display) -

This instruction subtracts one number represented in numeric display form from another.

This instruction has two numeric display operands. Its first operand is subtracted from its second operand and the result is left in the location of the second operand.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1   4 5    8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 is the number of digits in the first number; L2 is the number of digits in the second number.

S specifies the sign encoding of the result.

The first operand of this instruction can be an immediate operand. It is illegal for the second operand to be an immediate operand.

7.1.4 MOVND (Move Numeric Display) -

This instruction moves a numeric display field. If the destination length differs from the source length, the instruction truncates or zero fills on the left end.

This instruction has two operands whose addresses are represented in the form described in section 3. The first operand specifies the number to be moved and the second operand specifies its destination.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5     8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 and L2 are the lengths of the source and destination respectively.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

This instruction will never generate a result of minus 0. For example, if -1000 is moved from a 4 digit field to a 3 digit field a +000 result is generated.

It is illegal for the second operand of this instruction to be an immediate operand. It is legal for the first operand to be immediate.

7.1.5 ASHND (Arithmetic Shift Numeric Display) -

This instruction moves a numeric display field, and shifts the digits (9-bit bytes) to the right or left, truncating or supplying zero bytes on either end of the field. The shift amount, supplied in AC 0, specifies a byte displacement relative to the right most (low-order) byte of the source field.

This instruction has two operands whose addresses are respresented in the form described in section 3. The first operand specifies the number to be moved and shifted. The second operand specifies the destination.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1   4 5    8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 and L2 are the lengths of the source and destination respectively.

S specifies the sign encoding of the result.

The shift amount is given in AC 0 as a signed binary integer. A positive value causes a left shift or multiply by a power of 10. A negative value causes a right shift or divide by a power of 10. At the end of this instruction AC 0 is 0.

The shift is performed as if the length of the shift register were the longer of the lengths L1 and L2. The store is performed according to the rules specified for MOVND.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

The results of this instruction are unspecified if operands overlap or are identical to eachother.

It is illegal for the second operand of this instruction to be an immediate operand. It is legal for the first operand to be immediate.

7.1.6 TLGLND (Test For Legal Numeric Display) -

This instruction tests whether a character string contains a legal numeric display value and stores the test result in the flag word.

Its first operand specifies a 9-bit byte string the contents of which will be tested. This operand may be one or two words long and has the form described in section 3.

This operand is followed in the instruction stream by a word of the form:

```
 0 1  4 5  8 9  12  13  14 17 18                           35
--------------------------------------------------------------
!S! L1 !  0 !  0  !  0 !  0 !                              0 !
--------------------------------------------------------------
```

L1 is the number of digits in the number.

All 9-bit bytes except the low order byte must contain only numeric ASCII bytes (060 for 0 thru 071 for 9).

When the S bit is set to 1, the low order byte is allowed to have all legal sign encodings, including 'unsigned' encodings, ie, the ASCII bytes 0 to 9. When the S bit is set to 0, only the ASCII bytes 0 thru 9 are allowed in the low order byte.

If all bytes pass the above tests, bits 0, 1, and 2 of the flag word are set to 001. If any of the above conditions are not met, bits 0, 1, and 2 of the flag word are set to 000.

It is legal for the numeric display operand to be immediate.

7.1.7 CVTNDB (Convert Numeric Display To Binary) -

This instruction converts a numeric display to a binary integer. The result is stored in an AC pair.

The first operand of this instruction refers to a 9-bit byte string that contains the numeric display number to be converted. It has the form described in section 3 above and may be either one or two words long. It is followed in the instruction stream by a word of the form:

```
 0 1  4 5  8 9  12  13  14 17 18                             35
----------------------------------------------------------------
!S! L1 !  0 ! AC  ! 0  ! 0   !                               0 !
----------------------------------------------------------------
```

L1 is the number of digits in the first operand.

AC is an AC pair in which the binary form of the number is stored as a 10/20 format double integer. This AC must not be AC 0 or AC 17.

When S is 0, the result is "unsigned". In this case, the absolute value of the number is stored.

It is legal for the numeric display operand of this instruction to be immediate.

7.1.8 CVTBND (Convert Binary To Numeric Display) -

This instruction converts a binary integer to numeric display.

The first word of this instruction has the form:

```
 0 1  4 5  8 9  12  13  14 17 18                             35
----------------------------------------------------------------
!OPCODE      ! AC  !  0 !   0 !                              0 !
----------------------------------------------------------------
```

AC is the AC pair that contains the binary number to be converted to display. This AC must not be AC 0 or AC 17. After this instruction has been executed, AC and AC+1 will contain 0.

This word is followed in the instruction stream by an operand that refers to the 9-bit byte string in which the numeric display form of the number should be stored. This operand may be one or two words long and has the form described in section 3.

Bits 0-8 of this operand contain the fields:

```
         0 1    4 5    8
        -----------------
        !S!      !  L2  !
        -----------------
```

L2 is the number of bytes that should be stored. If the number does not fit in the number of bytes specified, high order decimal bytes are truncated.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is illegal for the numeric display operand of this instruction to be an immediate operand.

7.2 Packed Decimal Instructions

The packed decimal instructions operate on numbers stored in packed decimal format. These numbers are stored as 4 bit digits ("nibbles"), where two nibbles are stored right justified in each 9 bit byte. The rightmost nibble always contains the sign. All other nibbles but the rightmost nibble contain a binary integer in the range 0 to 9. The unused bit in each 9 bit byte must be zero. On output this bit will always be zero.

The addresses of packed decimal operands are represented in the form described in section 3. This address always refers to the leftmost nibble in a byte. There is no way to address the second nibble in a byte.

As with numeric display operations, those instructions that operate on two operands include 2 length fields, each of which is 4 bits long. Each length field contains the number of bytes in the corresponding operand. A length of 0 is interpreted as 16 bytes. Thus the instructions can operate on operands of different lengths, up to a maximum of 16 bytes (31 digits). Only numbers with an odd number of digits are supported.

As with numeric display operations, the S bit designates the form in which the algebraic sign is encoded in a result field. The sign encodings in hex are shown below:

hex	meaning	
0 to 9	illegal	
A	+	
B	-	
C	+	preferred plus sign
D	-	preferred minus sign
E	+	
F	+	preferred 'unsign' sign

The result of an arithmetic operation, a conversion operation, or a data movement operation contains either the preferred plus or the preferred minus sign when the S bit is set to 1. When the S bit is set to 0 the preferred 'unsign' sign is stored.

The source operands for these instructions can usually be immediate operands (see section 3). In an immediate operand, bits 18 to 35 of the effective address are treated as two 9-bit bytes. The first of these bytes contains 2 packed decimal digits. The second byte contains a digit in the left nibble and a sign encoding in the right nibble. Thus the immediate operand looks like a packed decimal number with byte length of 2, and can be in the range +999 to -999.

7.2.1 CMPP (Compare Packed) -

This instruction compares two numbers represented in packed decimal form. The result of the comparison is stored in the flag word (see 5.8). It treats plus and minus zero as identical.

This instruction has two packed decimal operands that specify the numbers to be compared.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5     8
-----------------
!0!  L1  !  L2  !
-----------------
```

L1 is the number of bytes in the first number; L2 is the number of bytes in the second number.

Either of the operands of this instruction can be immediate operands.

7.2.2 ADDP (Add Packed) -

This instruction adds two numbers represented in packed decimal form. The result is left in the location of the second operand.

This instruction has two packed decimal operands.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5     8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 is the number of bytes in the first number; L2 the number of bytes in the second number.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

The first operand of this instruction can be an immediate operand. It is illegal for the second operand to be an immediate operand.

7.2.3 SUBP (Subtract Packed) -

This instruction subtracts one number represented in packed decimal from another. Its first operand is subtracted from its second operand and the result is left in the location of the second operand.

This instruction has two packed decimal operands.

Bits 0-8 of the second operand of this instruction contain the fields:

```
  0 1    4 5     8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 and L2 are the number of bytes in each operand.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

The first operand of this instruction can be an immediate operand. It is illegal for the second operand to be an immediate operand.

7.2.4 MOVP (Move Packed) -

This instruction moves a packed decimal field. If the destination length differs from the source length, the instruction truncates or zero fills on the left end.

This instruction has two operands whose addresses are represented in the form described in section 3. The first operand specifies the number to be moved and the second operand specifies its destination.

Bits 0-8 of the second operand of this instruction contain the fields:

```
  0 1    4 5     8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 and L2 are the lengths of the source and destination respectively.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is illegal for the second operand of this instruction to be an immediate operand. It is legal for the first operand to be immediate.

7.2.5 ASHP (Arithmetic Shift Packed) -

This instruction moves a packed decimal field, and shifts the digits (nibbles) to the right or left, truncating or supplying zero digits on either end of the field. The shift amount, supplied in AC 0, specifies a digit displacement relative to the right most (low-order) digit of the source field. This instruction has two operands whose addresses are respresented in the form described in section 3. The first operand specifies the number to be moved and shifted. The second operand specifies the destination.

Bits 0-8 of the second operand of this instruction contain the fields:

```
         0 1    4 5     8
         -----------------
         !S!  L1  !  L2  !
         -----------------
```

L1 and L2 are the lengths of the source and destination respectively, in bytes of course.

The number of digits (not bytes!!!) to shift is given in AC 0 as a signed binary integer. A positive value causes a left shift or multiply by a power of 10. A negative value causes a right shift or divide by a power of 10. At the end of this instruction AC 0 is 0.

The shift is performed as if the length of the shift register were the longer of the lengths L1 and L2. The store is performed according to the rules specified for MOVP.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

The results of this instruction are unspecified if operands overlap or are identical to eachother.

It is illegal for the second operand of this instruction to be an immediate operand. It is legal for the first operand to be immediate.

7.2.6 TLGLP (Test For Legal Packed Decimal) -

This instruction tests whether a character string contains a legal packed decimal value and stores the test result in the flag word.

Its first operand specifies a 9-bit byte string the contents of which will be tested. This operand may be one or two words long and has the form described in section 3.

This operand is followed in the instruction stream by a word of the form:

```
 0 1  4 5  8 9  12  13  14 17 18                                 35
--------------------------------------------------------------------
!S! L1 !  0 !  0  !  0 !  0 !                                  0 !
--------------------------------------------------------------------
```

L1 is the number of bytes in the number.

All 9-bit bytes must contain:

```
 0 1  4 5  8
-------------
!0! N1 ! N2 !
-------------
```

In all bytes except the last N1 and N2 must be in the range 0 - 9. In the last byte, N1 must be in the range 0 - 9 and N2 must contain a legal sign. When the S bit is set to 1, this sign may be hex A - F. When the S bit is 0, this sign must be F.

If all bytes pass the above tests, bits 0, 1, and 2 of the flag word are set to 001. If any of the above conditions are not met, bits 0, 1, and 2 of the flag word are set to 000.

It is legal for the packed decimal operand to be immediate.

7.2.7 CHOP (Clear High Order Packed) -

This instruction clears the high order nibble (digit) of a packed decimal number. It is necessary because the packed decimal instructions only handle numbers that have an odd number of digits.

This instruction has a single operand which has the form described in section 3. This operand specifies the packed decimal number in which the high order digit will be cleared.

It is illegal for the operand of this instruction to be an immediate operand.

The result of this instruction will be -0 when the operand originally contained a negative value in which only the first nibble was non-zero. For example, CHOP of -100 gives a -0 result.

7.2.8 CVTPB (Convert Packed To Binary) -

This instruction converts a packed decimal to a binary integer.

The first operand of this instruction refers to a 9-bit byte string that contains the packed decimal number to be converted. It has the form described in section 3 above and may be either one or two words long. It is followed in the instruction stream by a word of the form:

```
 0 1  4 5  8 9  12  13  14 17 18                         35
-------------------------------------------------------------
!S! L1 !  0 ! AC  ! 0  ! 0    !                          0 !
-------------------------------------------------------------
```

L1 is the number of bytes in the number to be converted.

AC is an AC pair in which the binary form of the number is stored as a 10/20 format double integer. This AC must not be AC 0 or AC 17.

When S is 0, the result is "unsigned". In this case, the absolute value of the number is stored.

It is legal for the source operand of this instruction to be immediate.

7.2.9 CVTBP (Convert Binary To Packed) -

This instruction converts a binary number to packed decimal.

The first word of this instruction has the form:

```
 0 1  4 5  8 9  12  13  14 17 18                         35
-------------------------------------------------------------
!OPCODE      ! AC  !  0 !   0 !                          0 !
-------------------------------------------------------------
```

AC is the AC pair that contains the binary number to be converted to packed decimal. This AC must not be AC 0 or AC 17. After this instruction has been executed, AC and AC+1 will contain 0.

This word is followed in the instruction stream by an operand that refers to the 9-bit byte string in which the packed decimal form of the number should be stored. This operand may be one or two words long and has the form described in section 3.

Bits 0-8 of this operand contain the fields:

```
 0 1    4 5     8
------------------
!S! 0    !  L2  !
------------------
```

L2 is the number of bytes that will be stored.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is illegal for the destination operand of this instruction to be an immediate operand.

7.2.10 CVTNDP (Convert Numeric Display To Packed) -

This instruction converts a number represented in numeric display form (see section 6.1 above) to packed decimal.

It has two operands which both refer to 9-bit byte strings and have the form described in section 3.

The first operand refers to a 9-bit byte string that contains a number in numeric display form. The second operand refers to the destination in which this instruction will store that number in packed decimal form.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5     8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 is the number of digits in the numeric display field that is being converted. L2 is the number of bytes in the string in which it is being stored. If L1 is not equal to L2 * 2 - 1, left truncation or zero fill occurs.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is legal (if not useful) for the source operand to be an immediate operand. It is illegal for the destination operand to be an immediate operand.

7.2.11 CVTPND (Convert Packed To Numeric Display) -

This instruction converts a number from packed decimal form to numeric display.

It has two operands which both refer to 9-bit byte strings and have the form described in section 3.

The first operand refers to a 9-bit byte string that contains a number in packed decimal form. The second operand refers to the destination in which this instruction will store that number in numeric display form.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5    8
-----------------
!S!  L1  !  L2  !
-----------------
```

L1 is the number of bytes in the packed decimal string. L2 is the number of digits in the numeric display field. If L2 does not equal L1 * 2 - 1, left truncation or zero fill occurs.

S specifies the sign encoding of the result.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is legal (if not useful) for the source operand to be an immediate operand. It is illegal for the destination operand to be an immediate operand.

7.2.12 CVTEP (Convert EBCDIC Numeric Display To Packed) -

This instruction converts a number represented in EBCDIC characters to packed decimal.

The EBCDIC characters are stored as 8 bits right justified in a 9-bit byte. EBCDIC characters are divided into a four bit zone and a four bit digit. The packing operation consists of extracting only the digit bits from each EBCDIC byte and storing them into packed decimal nibbles.

The sign of the packed decimal field is controlled by the S bit.

When the S bit is 1, the zone bits of the low order EBCDIC byte are examined. If the zone bits contain a hex B or hex D, a hex D is stored in the packed decimal sign nibble. If the zone of the low order EBCDIC byte does not contain either a hex B or hex D, a hex C is stored in the packed decimal sign nibble.

When the S bit is 0, a hex F is stored in the packed decimal sign nibble, regardless of the value of the zone bits in the low order byte of the EBCDIC field.

This instruction has two operands which both refer to 9-bit byte strings and have the form described in section 3.

The first operand refers to a 9-bit byte string that contains a number in EBCDIC display form. The second operand refers to the destination in which this instruction will store that number in packed decimal form.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5     8
------------------
!S!  L1  !  L2  !
------------------
```

L1 is the number of digits in the EBCDIC field that is being converted. L2 is the number of bytes in the string in which it is being stored. If L1 is not equal to L2 * 2 - 1, left truncation or zero fill occurs.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is legal (if not useful) for the source operand to be an immediate operand. It is illegal for the destination operand to be an immediate operand.

7.2.13 CVTPE (Convert Packed To EBCDIC Numeric Display) -

This instruction converts a number from packed decimal form to EBCDIC form.

The packed decimal number is converted to EBCDIC as follows: Each packed decimal nibble except the sign nibble is moved to the 4-bit digit portion of an EBCDIC byte, and the 4-bit zone portion of the EBCDIC byte is supplied with a hex F. The zone portion of the low order EBCDIC byte is set with a sign as specified by the S bit.

When the S bit is 1, the packed decimal sign nibble is examined and if it contains a hex B or hex D, a hex D is stored in the zone bits of the low order EBCDIC byte. If the packed decimal sign nibble does not contain a hex B or hex D, a hex C is stored in the zone portion of the low order EBCDIC byte.

When the S bit is 0, the packed decimal sign nibble is ignored, and the zone portion of the low order EBCDIC byte is supplied with a hex F.

It has two operands which both refer to 9-bit byte strings and have the form described in section 3.

The first operand refers to a 9-bit byte string that contains a number in packed decimal form. The second operand refers to the destination in which this instruction will store that number in EBCDIC firm.

Bits 0-8 of the second operand of this instruction contain the fields:

```
 0 1    4 5     8
------------------
!S!  L1  !  L2  !
------------------
```

L1 is the number of bytes in the packed decimal string. L2 is the number of digits in the EBCDIC field. If L2 does not equal L1 * 2 - 1, left truncation or zero fill occurs.

This instruction compares the result as stored with 0 and stores the result of the comparison in the flag word.

It is legal (if not useful) for the source operand to be an immediate operand. It is illegal for the destination operand to be an immediate operand.

APPENDIX A

PERFORMANCE

The primary motivation for implementing the instructions described above is to gain substantial improvement in the performance of COBOL programs. A primary goal in specifying the formats of these instructions was their potential speed.

In almost all cases these instructions will be the fastest way to perform the operations that they implement. Software implementors will assume that the only exceptions to this are operations that can be performed by full word, half word, or double word move instructions.

APPENDIX B

EXTENSIONS TO BIS INSTRUCTIONS

The architecture committee has approved a number of extensions to the BIS instructions.

B.1 EXTENSIONS TO MOVST

The following extensions will be made to MOVST. These extensions will allow MOVST to be used as an efficient way to read records for stream format files.

1. Optional short translation table

   Bit 9 of E0 will be a flag bit that specifies that the translation table has entries only for characters 0 - 37. If this flag is 1, the translation for all characters greater than or equal to 40 is specified by bits 0 - 17 of E0+1.

   If this mode is specified, then function codes which specify character substitution (currently codes 0, 2, 3, 4, 6, 7) will result in characters greater than 40 being copied from source to destination (i.e. if a short table is used, the instruction never substitutes a different character for characters >= 40)

2. Optional fill

   Bit 10 of E0 will be a flag that specifies that no fill should be performed.

3. Expansion of the function code field.

   The function code field in each translation table entry will be 4 bits wide (bits 0 - 3 and bits 18 - 21). (Previously function codes were 3 bits wide, and there were 4 free bits to the right of each function code.)

   Documentation will describe function codes as 2 octal digits, in which only the left 4 bits are used. Thus the existing function codes, which were previously described by a single octal digit, will now be that digit followed by a zero. New function codes will all have 4 as their 2nd digit.

4. Function code for "terminate if S=1, ignore (delete) if S=0"

   Function code 14 will mean "terminate if S = 1, ignore (delete) if S = 0".

5. Function code for "ignore" (delete) character

   Function code 04 will mean "ignore (delete) this character"

6. Function code to set S only

   Function code 74 will mean either:

   1. Substitute the specified character and set S to 1 (if a long table is being used)

   2. Copy the source character and set S to 1 (for characters >= 40 when a short table is being used)

   Note that currently S cannot be set without also setting M or N.

Note that the expanded function code field and the new function codes will apply to the EDIT and CVTDBT instructions as well as to MOVST. The short translate table and optional fill options will not be implemented for EDIT and CVTDBT.

B.2 NEW EXTENDED COMPARE INSTRUCTIONS

Two new EXTEND instructions are added to the instruction set. They are Compare Strings with Offset (CMPSO) and Compare Strings with Translation (CMPST). These instructions allow comparison of byte strings from dissimilar code sets and dissimilar byte sizes without moving one or both fields in their entirety to temporary locations.

(The COBOL-79 compiler will have to provide the 6-bit and 7-bit byte strings that are provided in COBOL-68, as well as the 9-bit ASCII and EBCDIC byte strings that are needed for compatibility with IBM and VAX. Hence the potential exists for 2 or 3 dissimilar byte sizes, with different code sets existing side by side in the same object program.)

Each instruction has a single opcode value rather than a set of values corresponding to the 'EQ', 'NE', 'GT', 'LS', 'LE', and 'GE' forms of the existing CMPSxx instructions. The CMPSO instruction always takes a non-skip return.

The CMPST instruction normally takes a skip return; the non-skip return is taken if an abort condition is signalled in the translation of either operand.

The instructions return the result of the comparison as a condition code stored in three bits of the Flag word, and also stored in one of the ACs.

Both instructions deal with two operands, operand 1 being described by a length in AC, and a byte pointer in AC+1, and AC+2. Operand 2 is described by a length in AC+3, and a byte pointer in AC+4 and AC+5.

B.2.1 CMPSO Compare String With Offset

Use E1, the effective address of the CMPSO instruction, as an offset value for each byte of the first operand. The second operand is not offset. The bytes are compared from left to right on each string, with the shorter string extended by the fill byte stored at E0+1 or E0+2. The E0+1 fill byte is offset by E1. At the first inequality, the comparison stops, the byte pointers remain pointing at the bytes that were found unequal and the inequality condition is stored in the Flag word and in AC, see below. If no inequality is found, the instruction stops when the longer string is exhausted, and the equal condition is stored in the Flag word and in AC, see below.

| This instruction has EXTEND opcode 32.

B.2.2 CMPST Compare String With Translation

This instruction provides a compare facility with two translation tables, one for each of the operands.

| This instruction has EXTEND opcode 33.

E1, the effective address of the CMPST instruction, specifies the address of a block of three words. These three words have the form:

```
        0                17 18            35
        ------------------------------------
        !               TAB1               !
        ------------------------------------
        !               TAB2               !
        ------------------------------------
        !      FILL1     !      FILL2      !
        ------------------------------------
```

TAB1 and TAB2 are indirect words (IFIW or EFIW) that point to translation tables for string 1 and string 2 respectively. FILL1 and FILL2 are the fill characters to be used for string 1 and string 2 respectively.

If E1=0,	no translation is specified for either operand. The fill character to be used for both operands is contained in E0+1
If E1 NOT=0,	(E1) is the address of the translation table for operand 1 (E1+1) is the address of the translation table for operand 2 If either (E1)=0 or (E1+1)=0 no translation is involved for the respective operand.

The translated bytes are compared from left to right on each string, with the shorter string extended by the fill byte specified by FILL1 or FILL2, depending on whether the first or second operand is shorter. At the first inequality, the comparison stops, the byte pointers remain pointing at the bytes that were found unequal and the inequality condition is stored in the Flag word and in AC, see below. If no inequality is found, the instruction stops when the longer string is exhausted, and the equal condition is stored in the Flag word and in AC, see below.

The translation table format is the same as that of the MOVST instruction. In particular, if an operand is being translated, bytes from that operand are fetched, translated, and discarded until significance (S bit in AC or AC+3) comes on. S may be preset by the programmer before the instruction execution begins, or it may be turned on by the proper code pattern in the translation table. The fill byte is always considered significant, regardless of the setting of the appropriate S bit.

When an operand is not being translated, the S bit is ignored for that operand.

The translation table may also set N and M bits in AC or AC+3, as in MOVST.

Note that the new function codes proposed for MOVST in section A.1 above (codes to: "terminate if S = 1, ignore if S = 0", "ignore", "set S") will all be legal function codes for CMPST. The short translate table option and the no fill option that are proposed for

MOVST will not be available for CMPST.

The condition code for CMPSO and CMPST replaces the existing setting of bits 0, 1, and 2 of the Flag word as follows:

Bits 0,1,2	Meaning
100	operand 1 less than operand 2
010	operand 1 greater than operand 2
001	operand 1 equal to operand 2

Note that these bits correspond to the Overflow, Carry 0 and Carry 1 bits, and they are tested by the JFCL instruction. The JFCL encodings (with new mnemonics) are as follows:

```
JCCL     JFCL 10,     Jump on operand 1 < operand 2
JCCLE    JFCL 12,     Jump on operand 1 <= operand 2
JCCG     JFCL 4,      Jump on operand 1 > operand 2
JCCGE    JFCL 6,      Jump on operand 1 >= operand 2
JCCE     JFCL 2,      Jump on operand 1 = operand 2
JCCN     JFCL 14,     Jump on operand 1 not = operand 2
```

The same encodings are also placed into bits 0, 1, and 2 of AC. Here, they may be tested (without reset) by the following instructions.

```
JUMPL    Jump on operand 1 < operand 2
JUMPGE   Jump on operand 1 >= operand 2
TLNN     The choice of mask determines skip.
```

B.3 ADDITIONS TO THE EXTENDED EDIT INSTRUCTION.

Two new pattern bytes will be added to the EDIT instruction. Both new pattern bytes must be followed by a pattern byte which holds a count. The mnemonics are ADJDP, Adjust Destination Pointer and REPEAT, Repeat next pattern byte.

Also, the action of the edit instruction in adjusting the pattern byte number (PB#) is modified for all pattern bytes.

These new pattern codes will enable us to avoid codeing around some problems that are now handled in LIBOL. The same problems will exist in COBOL-79.

B.3.1 ADJDP (octal Code 005)

Interpret the next pattern byte as a twos complement number, sign extend it, and do the equivalent of an ADJBP on the destination pointer. This allows adjustment of the destination pointer in either direction.

B.3.2 REPEAT (octal Code 006)

Store the next pattern byte in bits 9 thru 17 of AC and increment PB#.

For all pattern codes except REPEAT, the incrementing of PB# is changed as follows:

After the action of the pattern code is complete, but before the incrementing of the pattern byte number, bits 9 to 17 of AC are examined. If they are non-zero, they are decremented, and the incrementation of the PB# is inhibited. If 9-17 of AC contain 0, PB# is incremented and the instruction proceeds as currently specified. The effect is to make all pattern codes execute the number of times stored in bits 9 to 17 of AC plus one.

In the case of STOP, ignore bits 9 thru 17 of AC, increment the PB#, and terminate the instruction execution, taking the skip return.

B.4 ADDITION TO THE ADJBP INSTRUCTION.

The ADJBP instruction will be extended so that it will expand a one word byte pointer (local form) into a two word byte pointer (global form) when the local byte pointer is fetched from across a section boundary. This is a general problem, not peculiar to COBOL.

If an ADJBP instruction is executed in a non-0 section it will deliver an extended byte pointer if the byte pointer is fetched from a section other than the section in which the instruction resides.

The instruction compares the section number of the PC to the section number of the effective address, and if they are unequal, it forces the byte pointer to extended format.

Caution: This makes the number of ACs affected by the instruction dependent on the relative locations of the instruction and the data. Furthermore, if the ADJBP is executed via an XCT instruction, the results many not be the same as the execution of the ADJBP itself.

APPENDIX C

ASSEMBLER NOTATION FOR CIS INSTRUCTIONS

This is a preliminary proposal for the assembler notation to be used for the CIS instructions.

C.1 GOALS

1. Relatively easy to write.
2. Relatively easy to read.
3. Relatively easy to put into the assembler.

C.2 HOW TO HANDLE MULTIPLE WORDS

Each opcode will generate only one word of the instruction. Succeeding words of the instruction will each be generated by writing an additional opcode or pseudo-op.

It is not desirable to have a macro with a large number of positional arguments since it would be very easy to get confused and most of the arguments won't be used in the common cases anyway.

C.3 HOW TO HANDLE NORMAL OPCODES VERSUS EXTENDS

Each of the new instruction's opcodes will be predefined in the assembler. If an instruction is an EXTEND instruction then the user will be required to write the instruction as a literal which is the object of the EXTEND opcode (i.e., "EXTEND [...]").

C.4 HOW TO GENERATE THE SECOND THROUGH NTH WORDS

As mentioned before each of the additional words of the long instructions begins with its own pseudo-op. The available pseudo-ops are:

1. .CILEN(n) - generates n in bits 0-8. AC and IXY are as in other instructions. n is assumed to be in the decimal radix. Examples:

   ```
   .CILEN(10) mode,adr
   ```

2. .CILN2(s/f,n1,n2) - used for decimal instructions. Generates s/f in bit 0, n1 in bits 1-4, and n2 in bits 5-8. All numbers are assumed to be in the decimal radix.

3. .CIRG2(s/f,r1,r2) - used for instructions which accept operand lengths in registers. Generates s/f in bit 0, r1 in bits 1-4, and r2 in bits 5-8. All numbers are assumed to in the current radix (i.e., last RADIX pseudo-op).

4. .CISIZ(n) - generates n in bits 1-12 and zero in bit 0. AC field may not be specified. IXY are specified as in other instructions. All numbers are in the decimal radix. Example:

   ```
   .CISIZ(10) @Y(X)
   ```

C.5 SPECIFYING MODE AND OFFSET

The assembler will have builtin symbols for each of the modes. Offsets are specified by adding an expression to the mode. This combination is then written as the AC field of the instruction. If the user ommiits the AC field (and the comma which follows it) then word aligned direct mode is assumed. The available symbols are:

1. .CIDIR - Direct mode. Add the Offset: .CIDIR+0, .CIDIR+1, .CIDIR+2, .CIDIR+3. If offset is not added then an offset of zero is assumed; this is normally done when the user wishes to think of the operand as an aligned string. Examples:

   ```
   MOVC adr
   MOVC .CIDIR,adr
   MOVC .CIDIR+0,adr
   MOVC .CIDIR+2,adr
   ```

2. .CIDEF - Deferred mode. Example:

   ```
   MOVC .CIDEF,adr
   ```

3. .CIIMM - Immediate mode. The immediate value is then written as the IXY field. Examples:

```
                MOVC .CIIMM,DC("A") ; String of "A"s.
                ADDND .CIIMM,DDC(-12) ; -12.
```

4. .CIDIS - Direct subscripted mode. The offset is written as in direct mode (.CIDIS, .CIDIS+0, .CIDIS+1, .CIDIS+2, .CIDIS+3). Examples:

```
                MOVC .CIDIS,adr
                MOVC .CIDIS+0,adr
                MOVC .CIDIS+3,adr
```

5. .CIDFS - Deferred subscripted mode. Example:

```
                MOVC .CIDFS,adr
```

C.6 SPECIFYING IMMEDIATE CONSTANTS

C.6.1 Immediate Display Constants

A macro, .CIDC, is provided which generates an immediate display character. It has a single argument which is an ASCII character code. Examples:

```
        MOVC            .CIIMM,.CIDC("A")
        MOVC            .CIIMM,.CIDC(.CHNUL)
        MOVC            .CIIMM,.CIDC(.CHTAB)
        MOVC            .CIIMM,.CIDC(" ")
```

C.6.2 Immediate Display Decimal Constants

A macro, .CIDDC, is provided which generates an immediate display decimal constant. The macro has two arguments, the first is the number (which may be signed), and the second is non-blank if the constant is to be treated as unsigned. The macro will take care of "overpunching" the sign and will generate the "preferred" sign. All values are in the decimal radix. Examples:

```
        ADDND           .CIIMM,.CIDDC(99,unsigned) ; Unsigned 99.
        ADDND           .CIIMM,.CIDDC(10)  ; +10.
        ADDND           .CIIMM,.CIDDC(-10) ; -10.
        ADDND           .CIIMM,.CIDDC(29) ; +29.
```

C.6.3 Immediate Packed Decimal Constants

A macro, .CIPDC, is provided which generates an immediate packed decimal constnat. The macro has two arguments, the first is the number and may be signed, the second is non-blank if the constant is to be unsigned. The macro will generate the "preferred" sign. All values are assumed to be in the decimal radix.

Examples:

```
        ADDP            .CIIMM,.CIPDC(99,unsigned) ; Unsigned 99.
        ADDP            .CIIMM,.CIPDC(1000)  ; +100.
        ADDP            .CIIMM,.CIPDC(-10)  ; -10.
        ADDP            .CIIMM,.CIPDC(29)  ; +29.
```

C.7 EXAMPLES

C.7.1 Simple MOVC

Moves 20 (decimal) characters located at word address "adr1" to word address "adr2".

```
        MOVC          .CIDIR,adr1
          .CILEN(20) .CIDIR,adr2
```

Since the default mode is direct mode with offset of zero, this case is better written as:

```
        MOVC          adr1
          .CILEN(20) adr2
```

C.7.2 Simple (but Unaligned) MOVC

Moves 20 (decimal) characters beginning at the third byte of adr1 to adr2 beginning at its second byte.

```
        MOVC          .CIDIR+2,adr1
          .CILEN(20) .CIDIR+1,adr2
```

C.7.3 MOVC With Deferred Source And Direct Subscripted Destination

Moves 15 (decimal) characters located at the address contained in adr1 to the ith element of the 15 character per element array based at adr2 where i is contained in adr3.

```
        MOVC          .CIDEF,adr1
          .CILEN(15) .CIDIS,adr2
          .CISIZ(15) adr3
```

C.7.4 MOVC with Direct Subscripted Source And Deferred Destination

Same as previous example, except that the operands are reversed.

```
        MOVC          .CIDIS,adr2
          .CISIZ(15) adr3
          .CILEN(15) DEF,adr1
```

C.7.5 MOVC With Both Operands Direct Subscripted

This case occurs when a COBOL user has the record descriptions:

```
        01 A OCCURS 100 TIMES.
                    02 A1 PIC XX.
                    02 A2 PIC XXX
```

and

```
        01 B OCCURS 200 TIMES.
                    02 B1 PIC XXXXX
                    02 B2 PIC XXXX
                    02 B3 PIC X
```

and

```
        01 I PIC 9(9) COMP.
        01 J PIC 9(9) COMP.
```

and attempts to perform the statement:

```
        MOVE B2(I) TO A2(J).
```

The code would be:

```
        MOVC          .CIDIS+1,B+1 ; MOVE B2(I)
          .CISIZ(10) I  ; . . .
          .CILEN(3) .CIDIS+2,A ; TO A2(I)
          .CISIZ(5) J  ; . . ..
```

C.7.6 Adding Two Packed Decimal Fields

The first operand is located beginning in the third byte of adr1 and is 2 bytes long. The second operand is located beginning in the second byte of adr2 and is 3 bytes long

```
        ADDP          .CIDIR+2,adr1
         .CILN2(1,2,3) .CIDIR+1,adr2
```

C.7.7 Simple Convert Packed To Numeric Display

The first operand is a 2 byte packed decimal number aligned at adr1. The second operand is a 5 character display decimal number aligned at adr2. The sign of the source field is to be ignored.

```
        EXTEND        [ CVTPND adr1
                         .CILN2(0,2,5) adr2 ]
```

C.7.8 Variable Length Character Move

```
        MOVE          R1,LENGTH1
        MOVE          R2,LENGTH2
        MOVCV         SOURCE
         .CIRG2(0,R1,R2) DEST
```

C.8 OPEN ISSUES

1. How to express other than immediate constants.
2. Better mnemonic names.
3. Names for the values of the s/f bit.

APPENDIX D

;REVISION HISTORY

REVISION HISTORY:

Revision	Changes
11	
	1. Changed opcode assignments for instructions that are EXTENDS. (There was a conflict with the opcode assignments for extended exponent FORTRAN).
	2. ASHP and ASHND do not work for identical source and destination (or overlapping source and destination).
	3. CHOP can generate a -0 result.
	4. AC 17 is an illegal operand for CVTBND, CVTNDB, CVTBP, CVTPB, since each of these instructions uses an AC pair and specification of AC 17 would result in AC 0 being used.
	5. CHOP of an immediate operand is illegal
	6. CVTBP and CVTBND clear the AC's that contained the source operand.
10	
	1. Fixed erroneous bit numbers for FXU, DCK
	2. Fixed typos
	3. Specified that new mode bits for MOVST also apply to CVTDBT and EDIT
9	

1. Specified opcode assignments.

2. Specified PXCT for character string manipulation instructions.

3. Specified results of overflow in decimal arithmetic operations

4. Specified results of MUUO's that occur during CIS instructions

5. CIS instructions will work in section 0

6. Changed M and O bit assignments for "immediate mode" operands

7. Added sign control option to CVTNDB and CVTPB

8. Specified MOVST extensions to improve performance when reading stream format files

9. Added TLGLP instruction

10. Reworded description of immediate mode operands

11. Added extensions to the EXTEND instructions which were approved in Sep-77 to appendix B

12. Added Jack Krupansky's proposal for assembler notation for these instructions as appendix C

13. Specified in the arithmetic shift instructions that the shift register has the length of the longer operand.

14. Deleted the "other suggestions" section. This section suggested that we add a "string search" instruction at some time in the future. MOVST can be used as a search instruction; and the short translate table option of MOVST makes it efficient for searching for characters < 40.

8

1. Modified the effective address calculation rules to use 5 different addressing modes: direct, deferred, direct subscripted, deferred subscripted, and immediate.

   Under this addressing scheme all operands that describe 9-bit byte strings may be either one or two words long and hence instructions may be 2,3 or 4 words long. The descriptions of the individual instructions were changed to reflect this.

   The LCG architecture committee and the COBOL-79 project members considered the problem of character addressing at great length and considered a number of different addressing schemes before agreeing on this one.

2. Made "immediate" into an operand mode and removed the "immediate" instructions (FILLC, CMPCI, CMPNDI, ADDNDI, CMPI, ADDPI).

3. Reorganized the first part of the document to include the goals, overview, addressing, and conventions sections.

7

1. Specified that when these instructions are imterrupted, the PC always points to the first word.

2. Added the interpretation of the instruction mnemonics

3. Specified that none of these instructions ever generate a -0 result.

4. Specified that the numeric display instructions always set the leftmost 5 bits in each byte of the result to legal ASCII (00011).

5. Specified that the shift amount for ASHND and ASHP will be specified in AC 0.

6. Specified the CHOP instruction.

7. Specified that the output from CVTPB and CVTNDB is a double integer.

8. Removed the FILLCV and CMPCVI instructions (their operations can be accomplished by using MOVCV and CMPCV with one of the operands equal to the null string)

9. Specified that the pad character to be used for MOVCV and CMPCV will be specified in AC 0 if bit 0 of the 2nd operand contains a 1. Otherwise the pad character is ASCII blank.

10. Changed the maximum string length supported by MOVCV and CMPCV to be the maximum number that can fit in 27 bits. This is necessary because we decided to use 9 of the bits in AC 0 to specify a pad character. This leaves only 27 bits for state information if an interrupt occurs.

11. Added an appendix that makes a general statement about the performance of these instructions

12. Updated the open issues list.

6

1. Specified that character string manipulation instructions will work for overlapping operands if the start of the destination precedes the start of the source. Specified that both character string and numeric operations will work if the 2 operands are identical.

2. Specified that in the character manipulation functions a length field of zero means that the operand is the null string.

3. Reworded the description of the contents of AC 0 at the end of a character string compare operation.

4. Specified the character codes to be used in numeric display operations.

5. Specified that -0 is always treated as identical to +0.

6. Changed the specification of immediate numeric operands to always use bits 18-35 of the I,X,Y calculation. Numeric display immediate operands are always 2 digits. Packed decimal immediate operands are always 3 digits. The length fields corresponding to immediate operands will always be 2 (rather than 0).

7. Changed the mnemonics used for immediate opcodes to end in "I"

8. Specified the variable length character string manipulation instructions.

9. Modified ASHND and ASHP so that a positive shift count means shift left; negative means shift right.

10. Removed the character set conversion instructions from this document. These instructions should be extend instructions and will be documented separately.

5

1. Specified that results are unspecified if operands overlap.
2. Changed mnemonics to be more like VAX and IBM.
3. Specified the formats of immediate mode operands.
4. Specified value in AC 0 when CMPC and CMPIC find equal strings.
5. Specified the result of overflow and illegal data in decimal instructions.
6. Specified a MOVSGN (move sign, numeric display) instruction.

7. Removed specific unsigned instructions, and generalized all decimal arithmetic instruction formats to include an S bit which governs the sign encoding in the result field.
8. Generalized all decimal arithmetic instructions to include two 4-bit length fields.
9. Specified the bits in the flag word that are set for various instructions. (Tested by JFCL).
10. Added a MOVP (move packed) instruction, which had been overlooked.
11. Added a ASHND and ASHP (arithmetic shift, numeric display and packed).
12. Added EPACK and EUPACK instructions, which convert between EBCDIC display format fields and packed decimal fields.
13. Specified operand formats for CVT79 and CVT97

4 1. Specified that interrupts will be handled by saving the state information in AC 0.
2. When a character string compare is performed, AC 0 is set to the character position of the first character where the strings differ.
3. At the end of the other instructions AC 0 is left set to 0.

3 1. Changed the conventions for overpunched signs to those proposed by Peter Conklin, Jeff Rudy, and Pat White in their memo of 31-May-78.
Changes were:
a. Generated signs for +0 to +9 become { and A to I rather than being 0 to 9. This is done for IBM compatibility.
b. The 026 keypunch representations of ? for +0 and : for -0 were put back in.
c. The 1401 representations of ? for +0 and ! for -0 were added.
2. Added unsigned operations: ADDUN, ADDUNI, SUBUN. These operations are necessary because the output values for the numeric display instructions are { to I for +0 to +9.

2 The length fields for numeric display and packed decimal were changed back to contain the length rather than length-1. Instead we will let a length of 0 mean 16 digits for numeric display; 16 bytes for packed decimal. This change was made because in cases where variable length items are being handled in the future it will be confusing and potentially time consuming to always have to subtract one from the length of an item.

1 1. The order of the operands for some of the instructions was changed so that the order is always: source, destination.
Instructions changed were: FILLC, ADDNDI, ADDND, SUBND, CVTBND, ADDIP, ADDP, SUBP, CVTBP
2. Length fields for packed decimal and numeric display instructions will contain length-1. Thus we can handle a maximum of 16 (rather than 15) digits for numeric display

and a maximum of 31 digits for packed decimal.
3. Sign conventions for numeric display were simplified. Use of ? and : for plus and minus zero (which were 0-26 keypunch conventions) were removed. Use of lower case letters for signed 1-9 were removed.
4. Explicitly stated that two wd instructions cannot lie across a section boundary, and that section boundaries can be crossed when an offset is added to a global base address.
5. The instructions to convert between binary and packed decimal and between binary and display will handle a 72 bit binary number in an AC pair. The micro-code will treat numbers that fit in a single AC (ie those of 12 digits or less) as a special case so that their conversion will not be slowed down by the use of 72 bits.
6. The section called "other suggestions" was added, to document some suggestions that have been made by other members of the languages group.