EK-KE44A-UG-002

KE44-A CISP User's Guide

Prepared by Educational Services
of
Digital Equipment Corporation

Printed in U.S.A.

This document was set on DIGITAL's DECset-8000 computerized typesetting system.

CONTENTS

FIGURES

CHAPTER 1
INTRODUCTION

1.1 PURPOSE AND SCOPE

This manual provides the data necessary for the installation and operation of the KE44-A Commercial Instruction Set Processor (CISP) option to the KD11-Z Central Processing Unit (CPU). The KE44-A option, which significantly extends the capability of the PDP-11/44 computer in the area of commercial data processing, is installed in the PDP-11/44 cabinet.

1.2 GENERAL DESCRIPTION

1.2.1 Commercial Instruction Set (CIS)

The commercial instruction set (CIS) is a series of instructions for manipulating byte strings to provide improved COBOL performance, text editing and word processing capability. The instruction set includes instructions for character handling and decimal string operations. Each of these instructions has two forms: register and in-line.

In the register form, descriptors are loaded into the general registers before the instruction is performed. With the in-line form, descriptors are accessed by descriptor address pointers. The CIS also includes "load-two" and "load-three" descriptor instructions that augment the register form. The op code for all CIS instruction is 076nnn.

1.2.2 The Microcode

The CIS instructions are implemented in microcode. The KE44-A microstore comprises 1,000 88-bit words. When a valid op code is received, the starting microstore address is entered and the instruction is performed; all of the microwords necessary to perform the op code specified operation are sequenced through. Each 88-bit microword is subdivided into 32 fields. The CPC field ⟨87:76⟩ of each microword is coded with the address of the next microword.

1.2.3 Hardware Description

The main hardware elements of the KE44-A are:

1. a Control Store board, and
2. a Data Path board.

The Control Store is a quad height M7091 board that contains the microcode in ROM form. The operational logic is on a hex-height M7092 board that contains four basic sections:

1. Instruction Register (IR) Decode, CIS Program Counter (CPC) and Microprocessor Counter (MPC) Addressing logic

2. Binary data path logic

3. Decimal data path logic

4. Status information and condition code generation logic

Chapter 3 is a block diagram description of these sections.

1.2.4 Suspension (Interrupts)

Since CIS instruction times may be long (due to large operands), a method is provided for giving system devices interrupt access to the processor. Thus, during CIS instructions a test is made at specific points in the microcode for Bus Request (BR) interrupts. If an interrupt is detected, the CIS instruction is automatically interrupted, i.e., "suspended", on a BR priority basis. During suspension, the CIS instruction is stopped and control is returned to the KD11-Z. The interrupt routine will then run, and one or more new CIS instructions can be executed during the period of suspension. At the end of this interrupt routine, control is returned to the KE44-A for completion of the suspended instruction. The entry point (microword address) for the suspended instruction is the same as the initial entry point. The control store contains a service interrupt save-state routine and a restore-from-service-interrupt routine.

1.3 RELATED HARDWARE MANUALS

The following hardware manuals are related to the KE44-A and may be purchased from Digital Equipment Corporation.

Title	Document Number	Availability
PDP-11/44 CP Subsystem Technical Manual	EK-KD11Z-TM	Hardcopy and Microfiche
PDP-11/44 System User's Guide	EK-11044-UG	Hardcopy
FP11-F Floating-Point Technical Manual	EK-FP11F-TM	Hardcopy and Microfiche

All purchase orders for hardware manuals should be forwarded to:

Digital Equipment Corporation
Accessory and Supplies Group (P086)
Cotton Road
Nashua, NH 03060

Purchase orders must show shipping and billing addresses and state whether a partial shipment will be accepted.

All correspondence and invoicing inquiries should be directed to the above address.

For information concerning microfiche libraries, contact:

Digital Equipment Corporation
Micropublishing Group BU/D2
12 Crosby Drive
Bedford, MA 01730

CHAPTER 2
INSTALLATION AND CHECKOUT

2.1 INSTALLATION

The two KE44-A modules plug into a dedicated 14-slot processor backplane. The M7091 control store module plugs into Sections C-F of slot 1; the M7092 data path module plugs into slot 2 (Figure 2-1). The M7091 module has no jumpers or switches for use in the field. The M7092 module, however, has one toggle switch (S1) whose lever is set toward the left (i.e., toward the center of the module) for normal operation (Figure 2-2).

NOTE

The lever of switch S1 is set to the right during manufacturing test only.

2.2 CHECKOUT

After installation, the KE44-A is checked out by running diagnostic CZKEEA (PDP-11 CIS Instruction Exerciser), which tests all CIS instructions in both register and in-line modes. Each instruction is tested under the following conditions:

1. Using all combinations of operand data types
2. In each of three processor modes (user, supervisor and kernel)
3. With memory management enabled and disabled
4. With D-space enabled and disabled
5. In an interrupt environment
6. For many cases of string length, string address, and string data.

ROWS

SLOTS	A B	C	D E F
1	M7090 (KD11-Z CIM)		M7091 (KE44–A)
2		M7092	(KE44–A)
3		M7093	(FP11–F)
4		M7094	(KD11–Z/DATA PATH)
5		M7095	(KD11–Z/CONTROL)
6		M7096	(KD11–Z/MFM)
7		M7097	(CACHE)
8		M7098	(KD11–Z/UBI)
9		M8722	(MS11–M)
10		M8722	(MS11–M)
11		M8722	(MS11–M)
12		M8722	(MS11–M)
13		SPC	
14	M9302, M9202, BC11–A		SPC

FRONT

NOTES

1. A G 727, G7270 CARD IS REQUIRED IN ROW D OF ANY UNUSED SPC SLOT TO PROVIDE BUS GRANT CONTINUITY
2. A G7273 CARD IS REQUIRED IN ROW C AND D OF ANY UNUSED SPC SLOT TO PROVIDE BUS GRANT CONTINUITY.
3. MODULES ARE INSERTED WITH COMPONENT SIDE TOWARD RIGHT SIDE OF BACKPLANE.

TK-4380

Figure 2-1 Module Placement in Processor Backplane

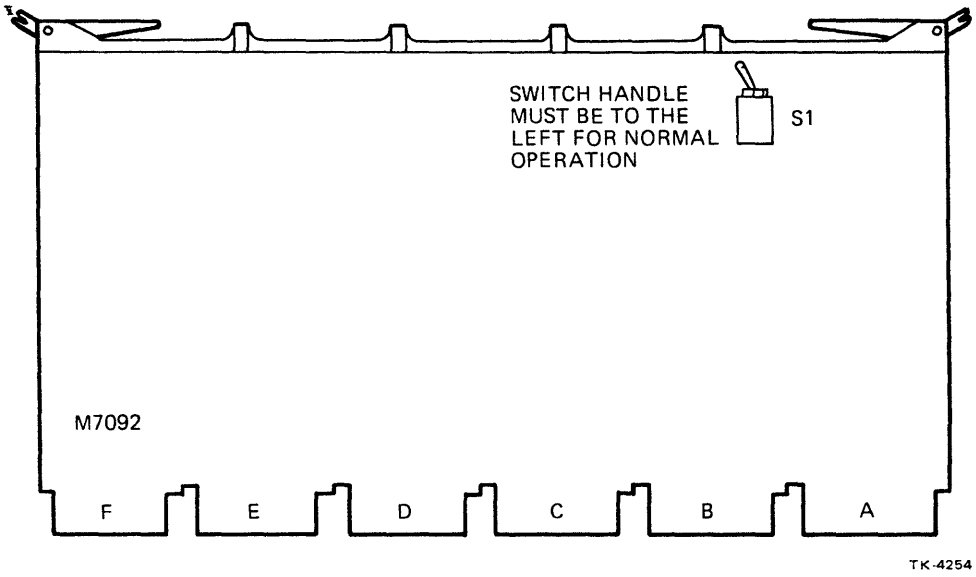


Figure 2-2 KE44-A Data Path/Logic Module, M7092

CHAPTER 3
BLOCK DIAGRAM DESCRIPTION

The KE44-A Commercial Instruction Set (CIS) comprises two modules. The M7091 is a quad module that contains the control store (microcode). The M7092 is a hex module that contains the operational logic. Refer to Figure 3-1.

Figure 3-1 is a block diagram of the KE44-A CIS, which has two basic operational modes: slave and master. The KE44-A is in a slave mode (idle state) until a valid CIS instruction is received. In this state, a given set of operations defined by the contents of the microword at control store address location 0000 is continually repeated. The mode of operation automatically transfers from slave to master when an op code in one of the following ranges is received:

076020 – 076077,
076120 – 076157, or
076170 – 076177

NOTE

In this manual, values shown in angle brackets (e.g., ⟨87:76⟩ indicate which bits of the control store word are being referenced.

When switched to master mode, the CIS takes control of the PDP-11/44 processor and begins to execute the CIS microcode.

The KE44-A outputs and inputs data to the KD11-Z via interconnecting AMUX lines. All memory references, whether input or output, are requested by the KE44-A, but are actually executed by the KD11-Z. The KE44-A controls the KD11-Z by setting appropriate bits in the control store, which then drive the KD11-Z microprocessor counter (MPC) lines. Data put on the AMUX lines by the KD11-Z is received by the CIS through its buffers, and distributed throughout the CIS by its internal MBUS.

When the operations specified by the CIS instruction are complete, the KE44-A returns to control store address 0000, (idle state), and control is returned to the KD11-Z.

As mentioned in the Introduction, there are four major functional areas within the KE44-A.

1. Instruction Register (IR) Decode, CIS Program Counter (CPC), and Microprocessor Counter (MPC) Addressing Logic.

2. Binary data path logic

3. Decimal data path logic

4. Status information and condition code generation logic

These functional areas are discussed in the following paragraphs.

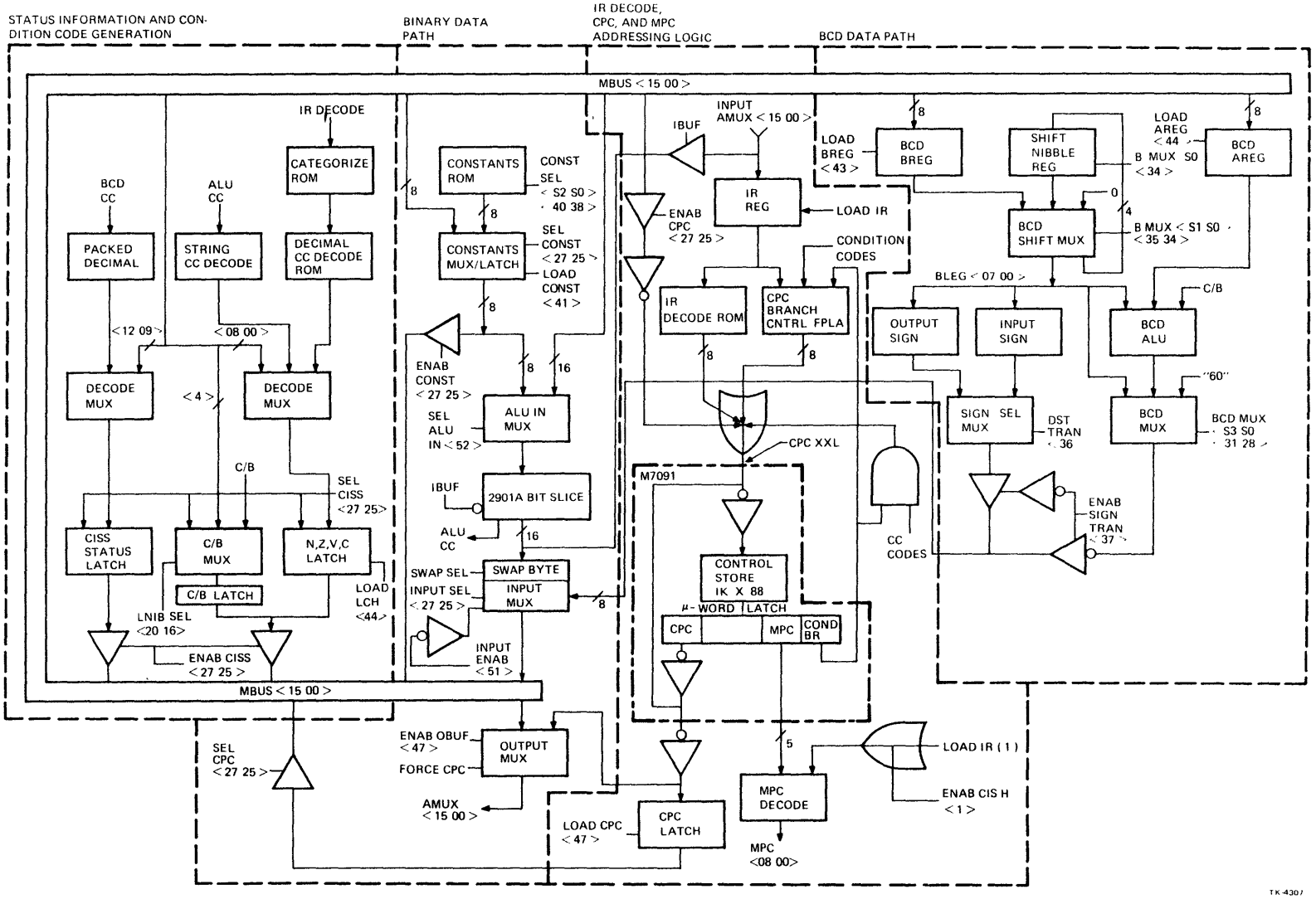


Figure 3-1 KE44-A Block Diagram

3.1 IR DECODE, CPC, AND MPC ADDRESSING

During the execution of non-CIS instructions, the KE44-A remains in its idle state, merely reading data from the AMUX lines. The AMUX lines are connected to the KE44-A via a set of line receivers that are inhibited when the IBUF H signal is zero (a one in ENIB ⟨48⟩).*

When the line receivers are enabled, the AMUX data is read to the "076" decode logic and the CIS instruction register. The instruction register is loaded by the KD11-Z signal LOAD IR L.

3.1.1 IR Register and IR Decode ROM

A CIS instruction is detected if a 0760xx or a 0761xx is present on the AMUX lines. The seven LSDs of the AMUX signal are inputs to the IR. IR output is applied to the starting CPC ROM, which outputs the starting CPC address. The starting CPC field ⟨87:76⟩ is the same for some instructions, e.g., the load word pair L2Dr and L3Dr, both of which result in a starting CPC of 161_8.

3.1.2 Control Store Module (M7091)

The starting CPC addresses the control store on the M7091, which outputs an 88-bit control word. At the beginning of the next clock cycle, this word is latched on the M7091 board and then used to direct and manipulate data on the M7092 data path board.

3.1.3 Field Programmable Logic Array (FPLA) for CPC Branch Control

A "next" CPC address can be modified if certain specific conditions are present. For example, if an overflow occurs, the CIS executes a set of microinstructions that differ from the normal set. The CIS address lines that can be modified to effect this change are CPC 02, CPC 01 and CPC 00.

Two enabling fields are provided to allow CPC address modification, i.e., branching. The CONBR1** field ⟨5:2⟩, if set, enables the FPLA**, thereby permitting CPC address modification to occur if specific conditions are set. The CONBR2 field ⟨9:6⟩, if set, enables a set of NAND gates that can also cause CPC addressing if another set of specific conditions exists.

One or more of the following conditions must exist if branching by the FPLA is to occur:

CIS IR 06 H
CIS IR 00 H
CIS IR 01 H
CIS NONZEROA H
CIS NONZEROB H
CIS NONZEROC H
CIS SRC1 ADR ODD H
CIS IR 04 H
CIS IR 05 H
CIS SIGN 1 H
CIS DST ADR ODD H
CIS SIGN 2 H

One or more of the following conditions must exist if branching by the NAND gates is to occur:

CIS PAGE FAULT H
CIS SUB OP H
CIS CCZ H
CIS CCC H
CIS SHFT OUT H
CIS CNN H
CIS C/B H
PFAIL BR PEND H

*See Appendix C for definition.
**See Appendix A for definition.

3.1.4 CIS Program Counter (CPC) Latch

The CPC latch is used for temporary storage of the CPC address. A return CPC address is usually stored in the CPC latch if a subroutine is called within the microcode.

The "next" CPC address is latched during a subroutine call by setting LOAD CPC ⟨42⟩. The latched address is used as a return address (at the end of the subroutine) by asserting CIS SEL CPC L ⟨27:25⟩.

3.1.5 MPC Decode

The MPC lines of the CIS are the same as the MPC lines of the KD11-Z processor. These lines are asserted by the CIS during a CIS instruction.

An MPC of 740_8 is asserted as soon as a 0760xx or 0761xx op code has been decoded. This prevents the KD11-Z processor from trapping to 10, since the processor itself does not recognize CIS instructions. The next MPC values are obtained from the microword MPC field (bits ⟨15:10⟩ "ORed" with 740_8). Therefore, during CIS operation, the MPC field reads 740_8 plus the value of MPC ⟨15:10⟩.

3.2 BINARY DATA PATH

3.2.1 Input Multiplexer and Swap Byte Logic

The binary data path operates on 16-bit data. The binary data path 2901A bit slice operates on the data selected from the MBUS or the constant ROM. MBUS data is received from the AMUX lines, read to the input multiplexer, and then passed straight through or swapped to the MBUS. The swap is controlled by bits ⟨50:49⟩ of the control store.

3.2.2 ALU Input Multiplexer

The ALU input multiplexer selects either the MBUS data or constants data. The ALU input multiplexer output is placed on the direct data input lines of the 2901A bit slice and stored in one of its 17 registers. MBUS data or constants data is selected by SALUI ⟨52⟩ of the control store.

3.2.3 2901A Bit Slice

The ALU, under the control of fields APORT ⟨75:72⟩, BPORT ⟨68:65⟩, ALUDST ⟨61:59⟩, ALUFTN ⟨58:56⟩, and ALUSRC ⟨55:53⟩, can perform three arithmetic and five logical operations.* The 2901A can also pass data between registers, and perform 16- or 32-bit left/right shifts. These shifts are accomplished by the 2901A and some additional circuitry. The results of these operations, if enabled, can output data to the input multiplexer and then to the MBUS.

3.2.4 Output Multiplexer and Maintenance Register

If data is needed by the KD11-Z, the MBUS data can be placed on the AMUX lines by setting ENAB OBUF ⟨47⟩. TRI-STATE AMUX L is also asserted by ENAB OBUF ⟨47⟩ and tri-states the KD11-Z lines, enabling CIS AMUX data to the KD11-Z.

The KD11-Z maintenance registers are used to view internal data. Three of these registers are used by CIS. They are:

E/M 1	–	CIS CPC address
E/M 2	–	CIS MBUS data
E/M 4	–	MPC address

Maintenance register 1 is enabled by having FREE BUS H and FORCE CPC L asserted from the M7096 module of the KD11-Z. These signals enable the output multiplexer and select the MBUS data for output.

*See Appendix C for mnemonic meanings.

Maintenance register 2 is enabled by having FREE BUS H and FORCE CIS DATA L asserted. These signals enable the output multiplexer and select the MBUS data for output.

Maintenance register 4 is enabled on the KD11-Z. It monitors the next MPC data and should be between 740_8 and 776_8 during a CIS instruction.

3.2.5 Constants ROM and Constants Multiplexer Latch

The binary data path contains a constants ROM. Constant selection is controlled by the CONST SEL field ⟨40:38⟩. The selected constant is fed to the constants multiplexer latch that outputs to the direct-data-in multiplexer of the 2901A.

3.3 BCD DECIMAL DATA PATH

The decimal data path operates on nibble data from the MBUS. Since a nibble is four bits (half a byte), there are two nibbles per byte. One nibble can easily represent the required range of binary coded decimal (BCD) numbers from zero to nine. The decimal operation is controlled by decode of a CIS instruction and the contents of the associated microword.

3.3.1 BCD "A" and "B" Registers

Two operands are needed in performing BCD nibble arithmetic. The two operands are transferred from the MBUS to the "A" register (BCD AREG) and the "B" register (BCD BREG), and then applied to the BCD ALU for calculation. The BCD AREG is loaded by asserting bit ⟨44⟩ and the BCD BREG by asserting bit ⟨43⟩ of the control store. The AREG data is applied directly to the BCD ALU, while the BREG data can be shifted before application to the BLD ALU.

3.3.2 BCD Shift Nibble Register and Shift Multiplexer

BREG data can be multiplied (left shift) or divided (right shift) by 10, or sent straight through to the BCD ALU.

The shift nibble register can be loaded with data during either a left or right shift. Ordinarily, this register is used to hold the sign data of an arithmetic string. These operations are controlled by the signals BMUX ⟨35:34⟩.

3.3.3 BCD ALU and Multiplier ROM

The BCD ALU is divided into two identical ROMs: high-nibble and low-nibble. BCD arithmetic is of a table look-up type; and the arithmetic operations performed by the ROMs are add, subtract, and multiply. Arithmetic operations are controlled by signals DEC 01 and DEC 00 that are derived from the IR, OP 01 ⟨33⟩, and OP 00 ⟨32⟩. The BCD arithmetic uses the AREG, BREG and control signals DEC 01 and DEC 00 as addresses to the BCD ALU ROM. The ROM output comprises 1) a result, 2) a carry, and 3) a status bit indicating whether or not the result is zero. The outputs of the arithmetic ROMs connect to the BCD MUX.

The largest result of a multiplication can be 9 * 9 = 81 which generates an answer of 1 and a carry of 8. All multiplication carries are generated by a separate multiply ROM.

3.3.4 BCD Output Multiplexer

The BCD MUX uses signals BCD MUX S3 through BCD MUX S0, ⟨31:28⟩, to select as an input either the BCD ALU ROM data, BREG data, a constant of 60, or a zero.

The BCD MUX output data is applied to the Input Multiplexer, whose output data goes on the MBUS.

3.3.5 Input/Output Sign Translators and Sign-Select Multiplexer

A second major operation performed in the decimal data path is sign translation. The input sign translator uses the sign of a packed or numeric format byte and, if necessary, changes it to the preferred sign format and extracts a BCD digit. (See Appendix A for sign values and location in the packed or numeric format.)

The output sign translator takes the sign bit of the result, outputs the preferred sign for the type of instruction involved and, if necessary, encodes a BCD digit with the sign.

Sign translation is enabled by asserting bit ⟨37⟩, whereas the input and output sign translators are enabled by deasserting bit ⟨36⟩.

3.4 STATUS INFORMATION

The CIS status word contains the condition codes (N, Z, V, and C) and the status bits for operations being performed by the KE44-A. The status bits are used to branch the CPC address to a different code sequence.

The status word, except for the condition code bits, is *not* available to the programmer by register access.

3.4.1 CIS Status Latch

The contents of the CIS status word are as follows:

Bit Function

0 – Carry (C)
1 – Overflow (V)
2 – Zero (Z)
3 – Negative (N)
4 – Carry/Borrow (C/B)
5 – NONZERO C
6 – NONZERO B
7 – NONZERO A
8 – SIGN 2
9 – SIGN 1
10 – DST ADR ODD
11 – SRC 2 ADR ODD
12 – SRC 1 ADR ODD
13 through 15 – No data, always zero.

3.4.1.1 Address Odd Bits – The three "address odd" bits indicate if either the source address (SRC 1 or SRC 2) and/or the destination address (DST ADR) are/is odd. These signals are enabled by the control store ⟨24:22⟩ and are latched to save the condition indicated.

The "address odd" bits can be tested at a later part of the microcode, and a CPC branch taken if these bits are set.

3.4.1.2 Sign Bits – Sign 1 and Sign 2 are the bits that indicate if the source address (SRC1 or SRC2) is negative. Both bits monitor the MBUS data via a multiplexer that selects certain bits depending on the data type of the CIS instruction being executed. Two signals (DAT TYPE 00 and DAT TYPE 01) are used to select data types. The following table shows the correspondence between data type, sign bits, and data select coding.

Data Type	Sign Bit	DAT TYPE 01	DAT TYPE 00
Character String	MBUS 15 or MBUS 07	0	0
Long Integer	MBUS 15	0	1
Arithmetic Zoned	MBUS 06	1	0
Arithmetic Packed	MBUS 00	1	1

These bits, enabled by control store bits ⟨20:16⟩, are latched to save the condition indicated by their respective states.

3.4.1.3 Non-Zero Bits – NONZERO A, NONZERO B and NONZERO C are individually latched to indicate the non-zero status of the BCD ALU. Each bit monitors either the high-nibble or low-nibble zero-status bit of the BCD ALU. These bits, each of which indicates a zero condition, are inverted to yield the NONZERO status bit. Each of the three latches holds one of the three zero conditions and is independently enabled by one of the control store bits ⟨20:16⟩.

3.4.2 Carry/Borrow Multiplexer and Latch

The Carry/Borrow (C/B) status bit indicates a Carry/Borrow in either the high-nibble or low-nibble of the BCD ALU. A multiplexer selects either the high- or low-nibble carry bit of the BCD ALU for storage in the C/B latch. The C/B latch is enabled by control store bit ⟨0⟩ and can also be forced by asserting FORCE C/B via control store bits ⟨20:16⟩.

3.5 CONDITION CODE GENERATION

3.5.1 Categorize ROM

The categorizing ROM that groups together instructions having similar condition codes, uses the seven least significant digits of the IR for its inputs. ROM outputs are used by the decimal CC decode ROM, the sign select multiplexer, and in the selection of either arithmetic or character condition codes.

3.5.2 String Character Condition Decode Logic

As indicated above, the character condition codes are selected by the categorize ROM. The settings of these codes are determined by the status signals from the 2901A bit slices. Low byte status is selected if the control store signal LOW BYTE H ⟨46⟩ is asserted. The condition codes and their associated 2901A signals are shown below.

	CCN	CCZ	CCC	CCV*
High Byte	MBUS 15	ALU 8-15 = 0	ALU COUT	See
Low Byte	MBUS 07	ALU 0-7 = 0	ALU COUT 7	below

3.5.3 Decimal Character Condition Decode Logic

Four outputs of the categorize ROM are used to partially address the decimal condition ROM. The status bits, NONZERO, SIGN and C/B are the remaining address bits. The addressed location, therefore, is based on the IR decode and the status bits of a decimal operation.

The decimal condition code ROM outputs the N, Z, V, C bits and a sign bit.

*$\overline{CCV}$ = (CCN (XOR) SIGN 2) . (SIGN 1 (XOR) SIGN 2)

3.5.4 N, Z, V, C Latch

The condition code bits are applied to the decode multiplexer which selects either character or decimal condition codes. The multiplexer output is applied to the N, Z, V, C latch that holds the condition code bits before their output to the MBUS. The data from the MBUS becomes an output to the AMUX and eventually to the PSW.

The N, Z, V, C latch is loaded by asserting bit ⟨45⟩.

NOTE

These Appendices have been duplicated directly from DECSTD168-PDP-11 Extended Instructions. Paragraphs 5.13 through 5.15 have been removed as they do not pertain to the KE44.

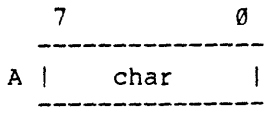
APPENDIX A
EXTENDED-INSTRUCTION DATA TYPES

3.1 CHARACTER DATA TYPES

There are three different character data types. The 'character' is a single byte, and is an abbreviated string of length one. The 'character string' is a contiguous group of bytes in memory. The third is a 'character set'.

3.1.1 Character

The character is an 8 bit byte:

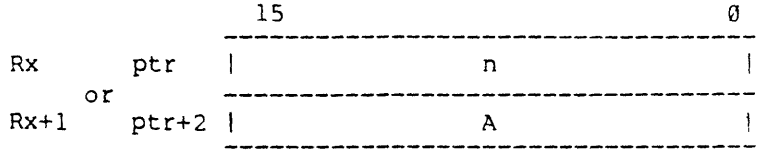


The character is used as an operand by CIS11 instructions. When it appears in a general register, the character is in the low order half; the high order half of the register must be zero. When it appears in the instruction-stream, the character is in the low order half of a word; the high order half of the word must be zero. If the high order half of a word which contains a character is non-zero, the effect of the instruction which uses it will be unpredictable.

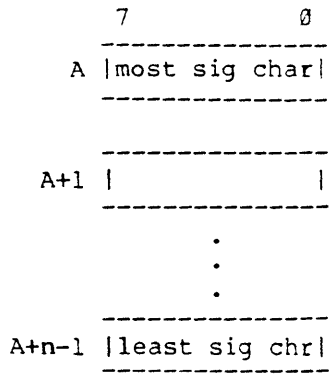
3.1.2 Character String

A character string is a contiguous sequence of bytes in memory that begins and ends on a byte boundary. It is addressed by its most significant character (lowest address). The highest address is the least significant character. It is specified by a two word descriptor with the attributes of length and lowest address. The length is an unsigned binary integer which represents the number of characters in the string and may range from Ø to 65,535. A character string with zero length is said to be vacant; its address is ignored. A character string with non-zero length is said to be occupied.

The character string descriptor is used as an operand by CIS11 instructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. The following figure shows the descriptor for a character string of length 'n' starting at address 'A' in memory:



The following figure shows the character string in memory:

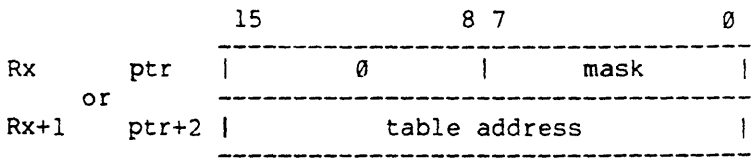


3.1.3 Character Set

A 'character set' is a subset of the 256 possible characters that can be encoded in a byte. It is specified by a descriptor which consists of the address of a 256 byte table and an 8 bit mask. The address is of the zeroeth byte in the table. Each byte in the table specifies up to eight orthogonal character subsets of which the corresponding character is a member. The mask selects which combinations of these orthogonal subsets comprise the entire character set. In effect, each bit in the mask corresponds to one of eight orthogonal subsets that may be encoded by the table. The mask specifies the union of the selected subsets into the character set. Typical sets would be: upper case, lower case, non-zero digits, end of line, etc.

Operationally, a character (char) is considered to be in the character set if the evaluation of (M[table.adr+char] AND mask) is not equal to zero. The character is not in the character set if the evaluation is zero. Each byte in the table indicates which combination of up to eight orthogonal character subsets (i.e. one for each of the eight bit vectors ØØØØØØØ1(2), ØØØØØØ1Ø(2), ØØØØØ1ØØ(2), ØØØ1ØØØØ(2), ØØ1ØØØØØ(2), Ø1ØØØØØØ(2) and 1ØØØØØØØ(2)) the corresponding character is a member. The mask specifies which union of the eight orthogonal character subsets comprise the total character set. For example, if the eight bit vector ØØØØØØØ1(2) appearing in the table corresponds to the character subset of all upper case alphabetic characters, ØØØØØØ1Ø(2) appearing in the table corresponds to the character subset of all lower case alphabetic characters, and ØØØØØ1ØØ(2) appearing in the table corresponds to the decimal digits, then using the mask ØØØØØØ11(2) with this table specifies the character set of all alphabetic characters, and using the mask ØØØØØ111(2) specifies the character set of all alphanumeric characters.

The character set descriptor is used as an operand by CIS11 instructions. It appears in two consecutive general registers, or in two consecutive words in memory pointed to by a word in the instruction stream. If the high order half of the first descriptor word is non-zero, the effect of an instruction which uses a character set will be unpredictable.



3.2 DECIMAL STRING DATA TYPES

Two classes of decimal string data types -- numeric strings and packed strings -- are defined. Both have similar arithmetic and operational properties; they primarily differ in the representation of signs and the placement of digits in memory.

The numeric string data types are signed zoned, unsigned zoned, trailing overpunch, leading overpunch, trailing separate and leading separate. The packed string data types are signed packed and unsigned packed. Instructions which operate on numeric strings permit each numeric string operand to be separately specified; similarly, packed string instructions permit each packed string operand to be separately specified. Thus, within each of the two classes of decimal strings, the operands of an instructions may be of any data type within the appropriate class.

3.2.1 Common Properties

Decimal strings exist in memory as contiguous bytes which begin and end on a byte boundary. They represent numbers consisting of Ø to 31(1Ø) digits in either sign-magnitude or absolute-value form. Sign-magnitude strings (SIGNED) may be positive or negative; absolute-value strings (UNSIGNED) represent the absolute value of the magnitude. Decimal numbers are whole integer values with an implied decimal radix point immediately beyond the least significant digit; they may be conceptually extended with zero digits beyond the most significant digit.

A 4-bit binary coded decimal representation is used for most digits in decimal strings. A four bit half byte is called a 'nibble' and may be used to contain a binary bit pattern which represents the value of a decimal digit. The following table shows the binary nibble contents associated with each decimal digit:

digit	nibble
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Each decimal string data type may have several representations. These representations permit certain latitude when accepting source operands. Decimal String data types have a PREFERRED representation which is a valid source representation and which is used to construct the destination string. Additional ALTERNATE representations are provided for some decimal data types when accepting source operands.

Decimal strings used as source operands will not be checked for validity. Instructions will produce upredictable results if a decimal string used as a source operand contains an invalid digit encoding, invalid sign designator, or in the case of overpunched numbers, an invalid sign/digit encoding.

When used as a source, decimal strings with zero magnitude are unique, regardless of sign. Thus, both positive and negative zero have identical interpretations.

Conceptually, decimal string instructions first determine the correct result, and then store the decimal string representation of the correct result in the destination string. A result of zero magnitude is considered to be positively signed. If the destination string can contain more digits than are significant in the result, the excess most significant destination string digits have zero digits stored in them. If the destination string can not contain all significant digits of the result, the excess most significant result digits are not stored; the instruction will indicate decimal overflow. Note that negative zero is stored in the destination string as a side effect of decimal overflow where the sign of the result is negative and the destination is not large enough to contain any non-zero digits of the result.

If the destination string has zero length, no result digits will be stored. The sign of the result will be stored in separate and packed strings, but not in zoned and overpunched strings. Decimal overflow will indicate a non-zero result.

3.2.2 Decimal String Descriptors

Decimal strings are represented by a two word descriptor. The descriptor contains the length, data type, and address of the string. It appears in two consecutive general registers (register form of instructions), or in two consecutive words in memory pointed to by a word in the instruction stream (in-line form of instructions). The unused bits are reserved by the architecture and must be Ø. The effect of an instruction using a descriptor will be unpredictable if any non-zero reserved fields in the descriptor contain non-zero values or a reserved data type encoding is used.

The design of the numeric and packed string descriptors are identical:

First Word:

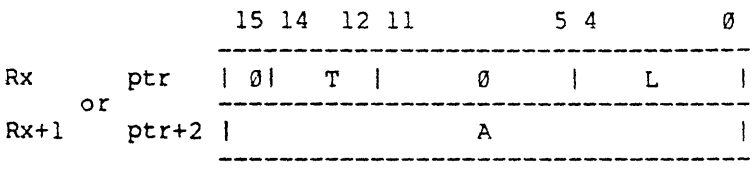
length <4:Ø> - Number of digits specified as an unsigned binary integer.

data type <14:12> - Specifies which decimal data type representation is used.

Second Word:

address <15:Ø> - Specifies the address of the byte which contains the most significant digit of the decimal string.

The following figure shows the descriptor for a decimal string of data type 'T' whose length is 'L' digits and whose most significant digit is at address 'A':



The encodings (in binary) for the NUMERIC string data type field are:

```
ØØØ  signed zoned
ØØ1  unsigned zoned
Ø1Ø  trailing overpunch
Ø11  leading overpunch
1ØØ  trailing separate
1Ø1  leading separate
11Ø  -- reserved by the architecture
111  -- reserved by the architecture
```

The encodings (in binary) for the PACKED string data type field are:

```
000  -- reserved by the architecture
001  -- reserved by the architecture
010  -- reserved by the architecture
011  -- reserved by the architecture
100  -- reserved by the architecture
101  -- reserved by the architecture
110  signed packed
111  unsigned packed
```

3.2.3 Packed Strings

Packed strings can store two decimal digits in each byte. The least significant (highest addressed) byte contains the the sign of the number in bits <3:0> and the least significant digit in bits <7:4>.

Signed Packed Strings -

The preferred positive sign designator is 1100(2); alternate positive sign designators are 1010(2), 1110(2) and 1111(2). The preferred negative sign designator is 1101(2); the alternate negative sign designator is 1011(2). Source strings will properly accept both the preferred and alternate designators; destination strings will be stored with the preferred designator.

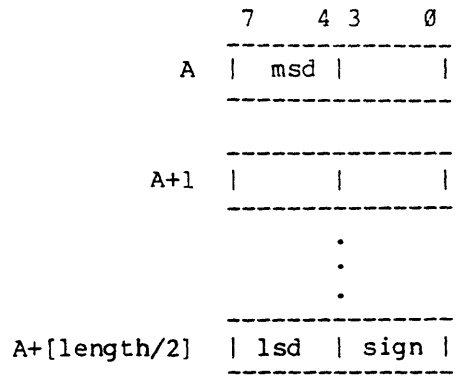
Unsigned Packed Strings -

PACKED SIGN NIBBLE:

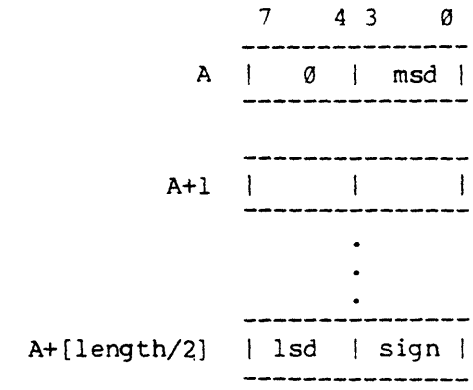
Sign Nibble	Preferred Designator	Alternate Designators
positive	1100(2)	1010(2) 1110(2) 1111(2)
negative	1101(2)	1011(2)
unsigned	1111(2)	

For other than the least significant byte, bytes contain two consecutive digits -- the one of lower significance in bits <3:0> and the one of higher significance in bits <7:4>. For numbers whose length is odd, the most significant digit is in bits <7:4> of the lowest addressed byte. Numbers with an even length have their most significant digit in bits <3:0> of the lowest addressed byte; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000(2) for destination strings. Numbers with a length of one occupy a single byte and contain their digit in bits <7:4>. The number of bytes which represent a packed string is [length/2]+1 (integer division where the fractional portion of the quotient is discarded).

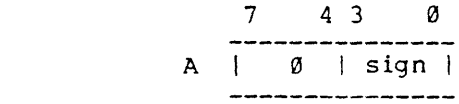
The following is a packed string with an odd number of digits:



The following is a packed string with an even number of digits:



A zero length packed string occupies a single byte of storage; bits <7:4> of this byte must be zero for source strings, and are cleared to 0000(2) for destination strings. Bits <3:0> must be a valid sign for source strings, and are used to store the sign of the result for destination strings. When used as a source, zero length strings represent operands with zero magnitude. When used as a destination, they can only reflect a result of zero magnitude without indicating overflow. The following is a zero length packed string:



A valid packed string is characterized by:

1. A length from 0 to 31(10) digits.

2. Every digit nibble is in the range 0000(2) to 1001(2).

3. For even length sources, bits <7:4> of the lowest addressed byte are 0000(2).

4. Signed Packed Strings - sign nibble is either 1010(2), 1011(2), 1100(2), 1101(2), 1110(2) or 1111(2).

5. Unsigned Packed Strings - sign nibble is 1111(2).

3.2.4 Zoned Strings

Zoned strings represent one decimal digit in each byte. Each byte is divided into two portions -- the high order nibble (bits <7:4>) and the low order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit.

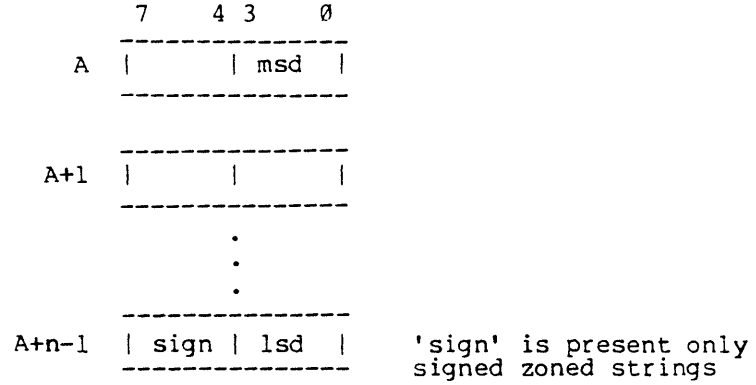
Signed Zoned Strings -

> When used as a source string, the high order nibble of the least significant byte contains the sign of the number; the high order nibbles of all other bytes are ignored. Destination strings are stored with the sign in the high order nibble of the least significant byte, and 0011(2) in the high order nibble of all other bytes. 0011(2) in the high order nibble corresponds to the ASCII encoding for numeric digits. The positive sign designator is 0011(2); the negative sign designator is 0111(2).

Unsigned Zoned Strings -

> When used as a source string, the high order nibbles of all bytes are ignored. Destination strings are stored with 0011(2) in the high order nibble of all bytes.

The number of bytes needed to contain a zoned string is identical to the length of the decimal number.



A zero length zoned string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a non-zero result will be indicated by setting overflow.

A valid zoned string is characterized by:

1. A length from Ø to 31(1Ø) digits.

2. The low order nibble of each byte is in the range ØØØØ(2) to 1ØØ1(2).

3. Signed Zoned Strings - The high order nibble of the least significant byte is either ØØ11(2) or Ø111(2).

3.2.5 Overpunch Strings

Overpunch strings represent one decimal digit in each byte. Trailing overpunch strings combine the encoding of the sign and the least significant digit; leading overpunch strings combine the encoding of the sign and the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions -- the high order nibble (bits <7:4>) and the low order nibble (bits <3:Ø>). The low order nibble contains the value of the corresponding decimal digit. When used as a source string, the high order nibble of all bytes which do not contain the sign are ignored. Destination strings are stored with ØØ11(2) in the high order nibble of all bytes which do not contain the sign. ØØ11(2) in the high order nibble corresponds to the ASCII encoding for numeric digits.

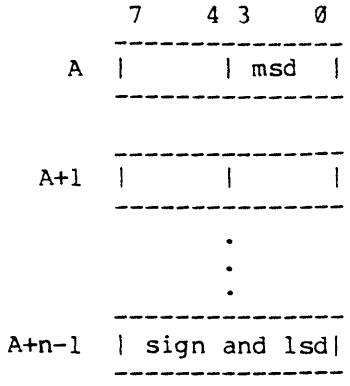
The following table shows the sign of the decimal string and the value of the digit which is encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics 'A' to 'R', '{' and '}'. The alternate designators correspond to the ASCII graphics 'Ø' to '9', '[', '?', ']', '!' and ':'.

OVERPUNCH SIGN/DIGIT BYTE:

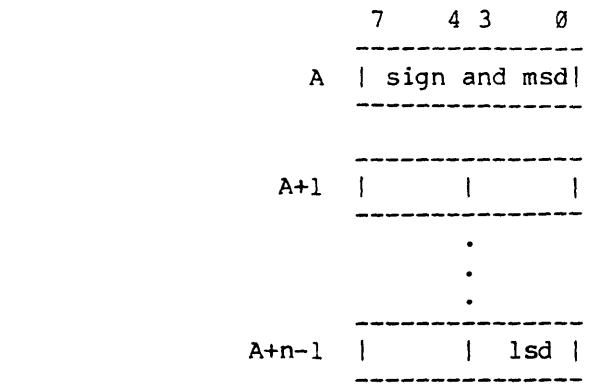
Overpunch Sign/Digit	Preferred Designator	Alternate Designators
+Ø	Ø1111Ø11(2)	ØØ11ØØØØ(2), Ø1Ø11Ø11(2), ØØ111111(2)
+1	Ø1ØØØØØ1(2)	ØØ11ØØØ1(2)
+2	Ø1ØØØØ1Ø(2)	ØØ11ØØ1Ø(2)
+3	Ø1ØØØØ11(2)	ØØ11ØØ11(2)
+4	Ø1ØØØ1ØØ(2)	ØØ11Ø1ØØ(2)
+5	Ø1ØØØ1Ø1(2)	ØØ11Ø1Ø1(2)
+6	Ø1ØØØ11Ø(2)	ØØ11Ø11Ø(2)
+7	Ø1ØØØ111(2)	ØØ11Ø111(2)
+8	Ø1ØØ1ØØØ(2)	ØØ111ØØØ(2)
+9	Ø1ØØ1ØØ1(2)	ØØ111ØØ1(2)
-Ø	Ø1111Ø1(2)	Ø1Ø111Ø1(2), ØØ1ØØØØ1(2), ØØ111Ø1Ø(2)
-1	Ø1ØØ1Ø1Ø(2)	
-2	Ø1ØØ1Ø11(2)	
-3	Ø1ØØ11ØØ(2)	
-4	Ø1ØØ11Ø1(2)	
-5	Ø1ØØ111Ø(2)	
-6	Ø1ØØ1111(2)	
-7	Ø1Ø1ØØØØ(2)	
-8	Ø1Ø1ØØØ1(2)	
-9	Ø1Ø1ØØ1Ø(2)	

The number of bytes needed to contain an overpunch string is identical to the length of the decimal number.

The following is a trailing overpunch string:



The following is a leading overpunch string:



A zero length overpunch string does not occupy memory; the address portion of its descriptor is ignored. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only accurately reflect a result of zero magnitude (the sign of the operation is lost). An attempt to store a non-zero result will be indicated by setting overflow.

A valid overpunch string is characterized by:

1. A length from Ø to 31(1Ø) digits.

2. The low order nibble of each digit byte is in the range ØØØØ(2) to 1ØØ1(2).

3. The encoded sign/digit byte contains values from the above table of preferred and alternate overpunch sign/digit values.

3.2.6 Separate Strings

Separate strings represent one decimal digit in each byte. Trailing separate strings encode the sign in a byte immediately beyond the least significant digit; leading separate strings encode the sign in a byte immediately beyond the most significant digit. Bytes other than the byte in which the sign is encoded are divided into two portions -- the high order nibble (bits <7:4>) and the low order nibble (bits <3:Ø>). The low order nibble contains the value of the corresponding decimal digit.

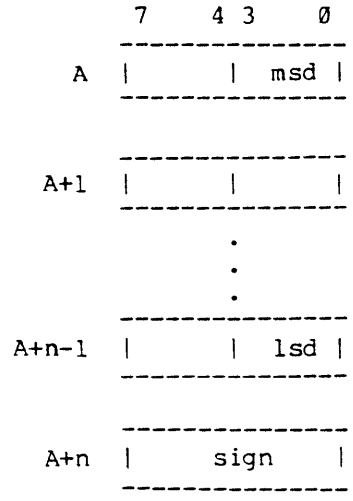
When used as a source string the high order nibbles of all digit bytes are ignored. Destination strings are stored with 0011(2) in the high order nibble of all digit bytes. 0011(2) in the high order nibble corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 00101011(2) and the alternate positive sign designator is 00100000(2). The negative sign designator is 00101101(2). These designators correspond to the ASCII encoding for '+', 'space' and '-'.

SEPARATE SIGN BYTE:

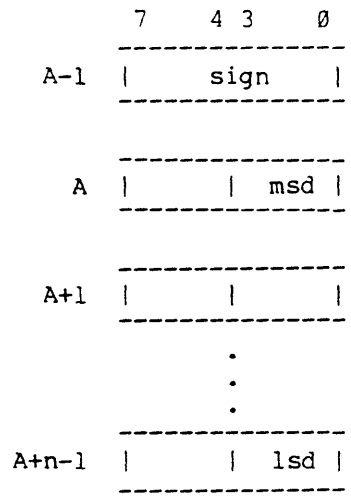
Sign Byte	Preferred Designator	Alternate Designators
positive	00101011(2)	00100000(2)
negative	00101101(2)	

The number of bytes needed to contain a leading or trailing separate string is identical to length+1.

The following is a trailing separate string:

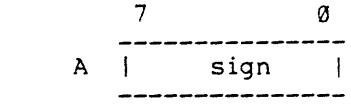


The following is a leading separate string:

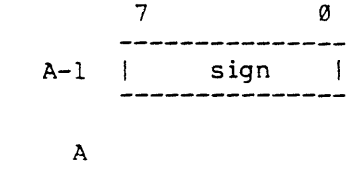


A zero length separate string occupies a single byte of memory which contains the sign. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow; the sign of the result is stored.

The following is a zero length trailing separate string:



The following is a zero length leading separate string:



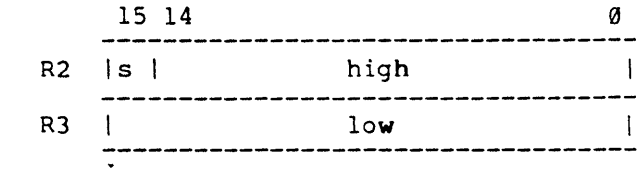
A valid separate string is characterized by:

1. A length from Ø to 31(1Ø) digits.
2. The low order nibble of each digit byte is in the range ØØØØ(2) to 1ØØ1(2).
3. The sign byte is either ØØ1ØØØØØ(2), ØØ1Ø1Ø11(2) or ØØ1Ø111Ø1(2).

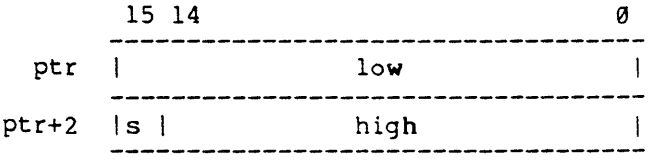
3.3 LONG INTEGER

Long integers are 32 bit binary two's complement numbers organized as two words in consecutive registers or in memory -- no descriptor is used. One word contains the high order 15 bits. The sign is in bit<15>; bit<14> is the most significant. The other word contains the low order 16 bits with bit<Ø> the least significant. The range of numbers that can be represented is -2,147,483,648 to +2,147,483,647.

The register form of decimal convert instructions use a restricted form of long integer with the number in the general register pair R2-R3:



The in-line form of decimal convert instructions reference the long integer by a word address pointer which is part of the instruction stream:

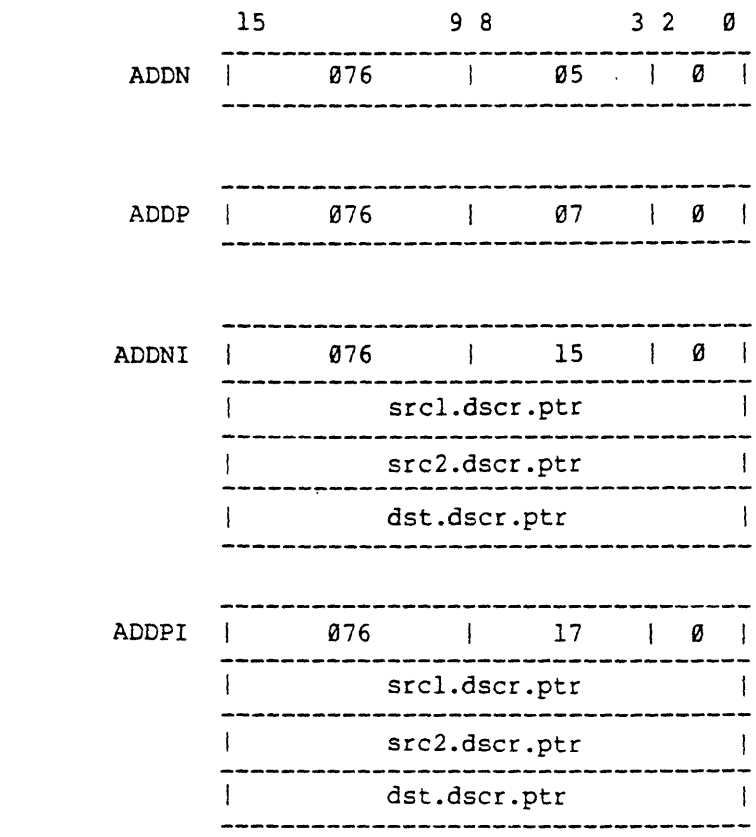


Note that these two representations of long integers differ. There is no single representation of long integer among EAE, EIS, FPP and software. The "register form" was selected to be compatible with EIS; the "in-line form" was selected to be compatible with current standard software usage.

APPENDIX B
EXTENDED-INSTRUCTION DEFINITIONS

5.1 ADDN / ADDP / ADDNI / ADDPI - Add Decimal

Format:



Operation:

```
dst <- src2 + src1
```

Condition Codes:

```
N:  set if dst<0;  cleared otherwise
Z:  set if dst=0;  cleared otherwise
V:  set if dst can not contain all significant digits of the
    result;  cleared otherwise
C:  cleared
```

Suspendability:

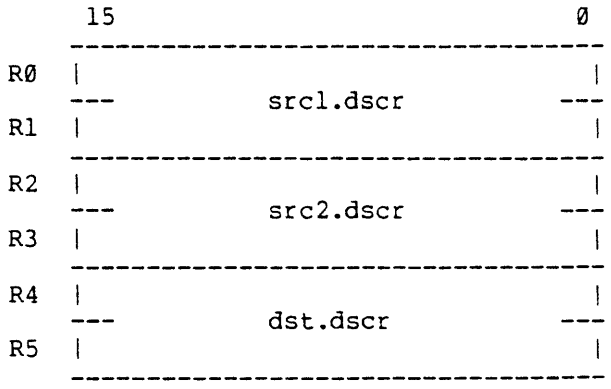
This instruction is potentially suspendable.

Description:

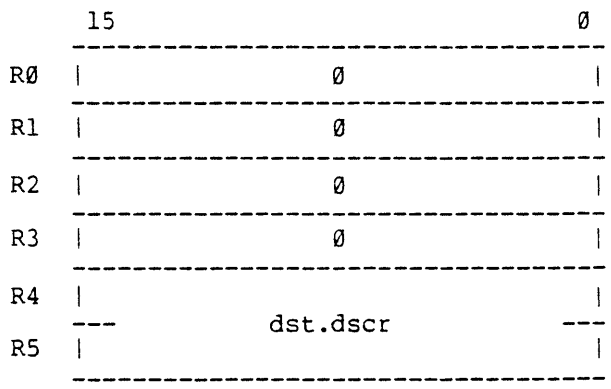
Srcl is added to src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - ADDN and ADDP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form - ADDNI and ADDPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Three Address Add - Register Form

```
        MOV     SRC1.DSCR,R0    ; 1st source descriptor
        MOV     SRC1.DSCR+2,R1
        MOV     SRC2.DSCR,R2    ; 2nd source descriptor
        MOV     SRC2.DSCR+2,R3
        MOV     DST.DSCR,R4     ; destination descriptor
        MOV     DST.DSCR+2,R5
        ADDN / ADDP             ; add
        BVS     OVERFLOW        ; check for error
        BLT     NEGATIVE        ; negative destination
        BEQ     EQUAL           ; zero destination
        BGT     GREATER         ; positive destination
```

2. Three Address Add - In-line Form

```
        ADDNI / ADDPI           ; add
        .WORD   SRC1.DSCR.PTR   ; ptr to src1 descriptor
        .WORD   SRC2.DSCR.PTR   ; ptr to src2 descriptor
        .WORD   DST.DSCR.PTR    ; ptr to dst descriptor
        BVS     OVERFLOW        ; check for error
        BLT     NEGATIVE        ; negative destination
        BEQ     EQUAL           ; zero destination
        BGT     GREATER         ; positive destination
```

3. Two Address Add - Register Form

```
        MOV     SRC.DSCR,R0     ; source descriptor
        MOV     SRC.DSCR+2,R1
        MOV     DST.DSCR,R2     ; destination descriptor
        MOV     DST.DSCR+2,R3
        MOV     R2,R4           ; duplicate destination
        MOV     R3,R5
        ADDN / ADDP             ; add
        BVS     OVERFLOW        ; check for error
        BLT     NEGATIVE        ; negative destination
        BEQ     EQUAL           ; zero destination
        BGT     GREATER         ; positive destination
```

4. Two Address Add - In-Line Form

```
ADDNI / ADDPI           ; add
.WORD   SRC.DSCR.PTR    ; ptr to src descriptor
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

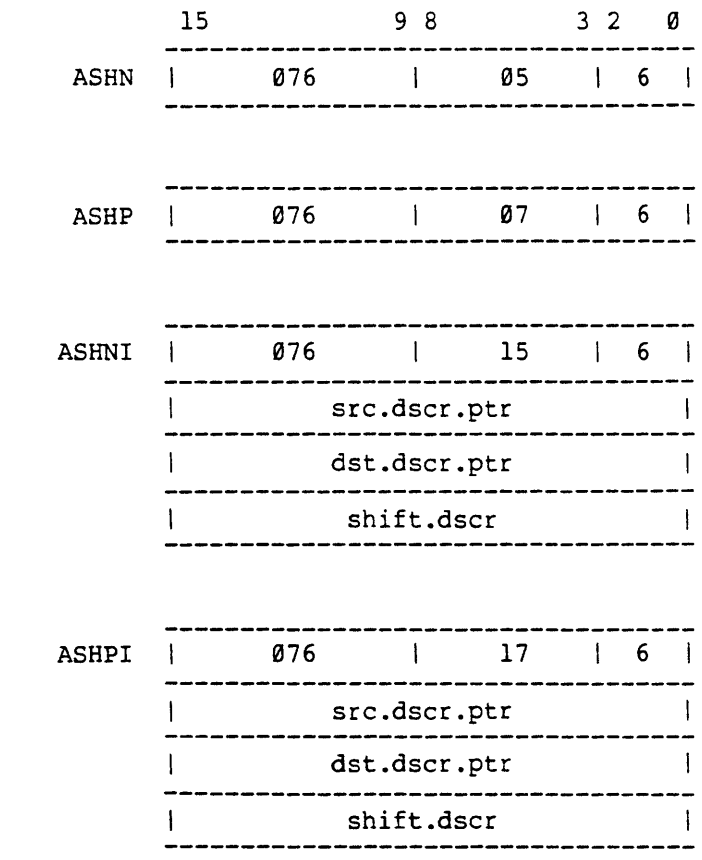
Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

5.2 ASHN / ASHP / ASHNI / ASHPI - Arithmetic Shift Decimal

Format:



Operation:

```
dst <- src * (10 ** shift count)
```

Condition Codes:

```
N:  set if dst<0;  cleared otherwise
Z:  set if dst=0;  cleared otherwise
V:  set if dst can not contain all significant digits of the
    result; cleared otherwise
C:  cleared
```

Suspendability:

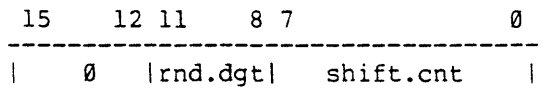
This instruction is potentially suspendable.

Description:

The decimal number specified by the source descriptor is arithmeticly shifted, and stored in the area specified by the destination descriptor. The shifted result is aligned with the least significant digit position in the destination string. The shift count is a two's complement byte whose value ranges from -128(10) to +127(10). If the shift count is positive, a shift in the direction of least to most significant digits is performed. A negative shift count performs a shift from most to least significant digit. Thus, the shift count is the power of ten by which the source is multiplied; negative powers of ten effectively divide. Zero digits are supplied for vacated digit positions. A zero shift count will move the source to the destination. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

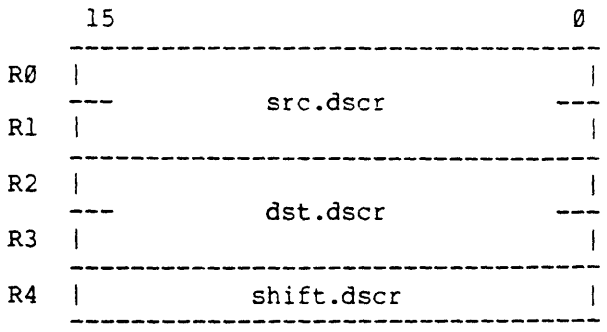
A negative shift count invokes a rounding operation. The result is constructed by shifting the source the specified number of digit positions. The rounding digit is then added to the most significant digit which was shifted out. If this sum is less than 10(10), the shifted result is stored in the destination string. If the sum is 10(10) or greater, the magnitude of the shifted result is increased by 1 and then stored in the destination string. If no rounding is desired, the rounding digit should be zero.

The shift count and rounding digit are represented in a single word referred to as the shift descriptor. Bits <15:12> of this word must be zero:

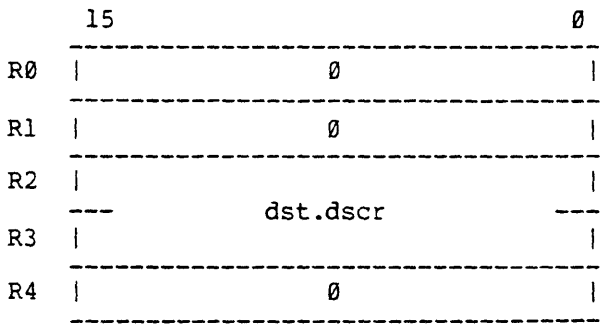


Register Form - ASHN and ASHP

When the instruction starts, the operands must have been placed in the general registers. The source descriptor is placed in R0-R1, the destination descriptor is placed in R2-R3, and the shift descriptor is placed in R4:



When the instruction is completed, the source descriptor registers and shift descriptor register are cleared:



In-line Form - ASHNI and ASHPI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word decimal string source descriptor, a word address pointer to a two word decimal string destination descriptor, and a shift descriptor word. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Multipling by 100 - Register Form

```
MOV     SRC.DSCR,R0     ; source descriptor
MOV     SRC.DSCR+2,R1
MOV     DST.DSCR,R2     ; destination descriptor
MOV     DST.DSCR+2,R3
MOV     #2,R4           ; shift descriptor word
ASHN / ASHP             ; shift
```

```
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

2. Multipling by 100 - In-line Form

```
ASHNI / ASHPI           ; shift
.WORD   SRC.DSCR.PTR    ; ptr to src descriptor
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
.WORD   2               ; shift descriptor word
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

3. Move decimal number - Register Form

```
MOV     SRC.DSCR,R0     ; source descriptor
MOV     SRC.DSCR+2,R1
MOV     DST.DSCR,R2     ; destination descriptor
MOV     DST.DSCR+2,R3
CLR     R4              ; shift descriptor word
ASHN / ASHP             ; shift
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

4. Move decimal number - In-line Form

```
ASHNI / ASHPI           ; shift
.WORD   SRC.DSCR.PTR    ; ptr to src descriptor
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
.WORD   0               ; shift descriptor word
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

Notes:

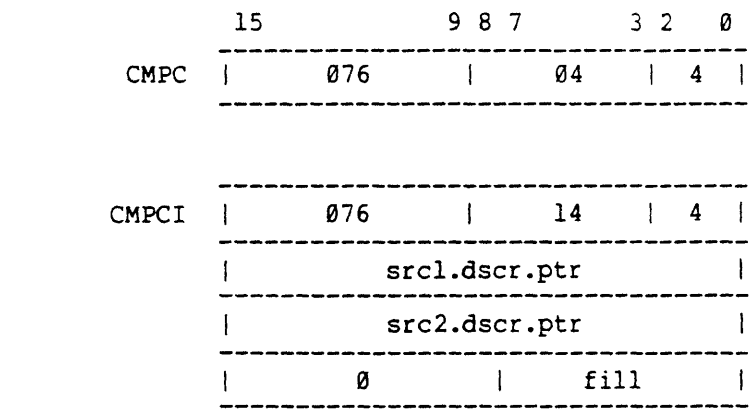
1. If bits <15:12> of the shift descriptor word are not zero, the effect of the instruction is unpredictable.

2. If bits <11:8> of the shift descriptor are not a valid decimal digit, the results of the instruction are unpredictable.

3. Any overlap of the source and destination strings will produce unpredictable results.

5.3 CMPC / CMPCI - Compare Character

Format:



Operation:

Srcl is compared with src2 (srcl-src2).

Condition Codes:

The condition codes are based on the arithmetic comparison of the most significant pair of unequal srcl and src2 characters (srcl.byte-src2.byte).

N: set if result<0; cleared otherwise
Z: set if result=0; cleared otherwise
V: set if there was arithmetic overflow, that is, srcl.byte<7> and src2.byte<7> were different, and src2.byte<7> was the same as bit <7> of (srcl.byte-src2.byte); cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Suspendability:

This instruction is potentially suspendable.

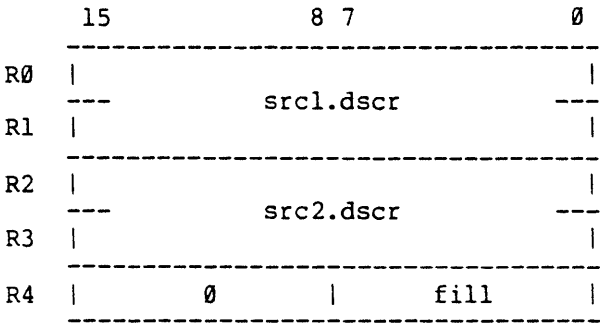
Description:

Each character of srcl is compared with the corresponding character of src2 by examining the character strings from most significant to least significant characters. If the character strings are of unequal length, the shorter character string is conceptually extended to the length of the longer character string with fill characters beyond its least significant character. The instruction terminates when the first corresponding unequal characters are found or when both character strings are exhausted.

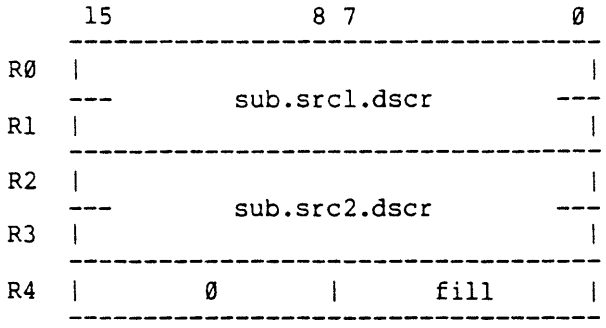
The condition codes reflect the last comparison, permitting the unsigned branch instructions to test the result.

Register Form - CMPC
-------- ---- - ----

When the instruction starts, the operands must have been placed in the general registers. The first source character string descriptor is placed in RØ-Rl, the second source character string descriptor is placed in R2-R3, the fill character is placed in R4<7:Ø>, and R4<15:8> must be zero:



The instruction terminates with sub-string descriptors in RØ-Rl and R2-R3 which represent the portion of each source character string beginning with the most significant corresponding unequal characters. RØ-Rl contain a descriptor for the unequal portion of the original srcl string; R2-R3 contain a descriptor for the unequal portion of the original src2 string. A vacant character string descriptor indicates that the entire source character string was equal to the corresponding portion of the other source character string, including extension by the fill character; its address is one greater than that of the least significant character of the character string.



In-line Form - CMPCI
------- ---- - -----

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string src1 descriptor, a word address pointer to a two word character string src2 descriptor, and a word whose low order half contains the fill character and whose high order half must be zero. R0-R6 are unchanged when the instruction is completed.

Formal Description:

```
src1.len = R0;          ! CMPC only
src1.adr = R1;          !    .
src2.len = R2;          !    .
src2.adr = R3;          !    .
fill = R4<7:0>;         !    .

temp = M[R7];           ! CMPCI only
src1.len = M[temp];     !    .
src1.adr = M[temp+2];   !    .
R7 = R7+2;              !    .
temp = M[R7];           !    .
src2.len = M[temp];     !    .
src2.adr = M[temp+2];   !    .
R7 = R7+2;              !    .
fill = M[R7]<7:0>;      !    .
R7 = R7+2;              !    .

found = 1;
while (src1.len nequ 0) and (src2.len nequ 0)
         and (found nequ 0) do
         if (M[src1.adr] eqlu M[src2.adr]) then
             begin
             src1.len = src1.len-1;
             src1.adr = src1.adr+1;
             src2.len = src2.len-1;
             src2.adr = src2.adr+1
             end
         else found = 0;
while (src1.len nequ 0) and (found nequ 0) do
         if M[src1.adr] eqlu fill then
             begin
             src1.len = src1.len-1;
             src1.adr = src1.adr+1
             end
         else found = 0;
while (src2.len nequ 0) and (found nequ 0) do
         if M[src2.adr] eqlu fill then
             begin
             src2.len = src2.len-1;
```

```
            src2.adr = src2.adr+1
            end
         else found = 0;

    if (src1.len eqlu 0) then btmp1 = fill
            else btmp1 = M[src1.adr];
    if (src2.len eqlu 0) then btmp2 = fill
            else btmp2 = M[src2.adr];
    carry@btmp = btmp1-btmp2;
    N = btmp<15>;
    if btmp eql 0 then Z = 1 else Z = 0;
    if (btmp1<7> neq btmp2<7>) and (btmp2<7> eql btmp<7>) then
            V = 1 else V = 0;
    C = carry;

    R0 = src1.len;          ! CMPC only
    R1 = src1.adr;          !    .
    R2 = src2.len;          !    .
    R3 = src2.adr;          !    .
    R4 = 0<15:8>@fill;      !    .
```

Examples:

1. Compare Strings - Register Form

```
        MOV     SRC1.DSCR,R0    ; 1st source descriptor
        MOV     SRC1.DSCR+2,R1
        MOV     SRC2.DSCR,R2    ; 2nd source descriptor
        MOV     SRC2.DSCR+2,R3
        MOV     #' ,R4          ; extend with spaces
        CMPC                    ; compare
        BLO     LESS            ; src1<src2
        BEQ     EQUAL           ; src1=src2
        BHI     GREATER         ; src1>src2
```

2. Compare Strings - In-line Form

```
        CMPCI                   ; compare
        .WORD   SRC1.DSCR.PTR   ; ptr to src1 descriptor
        .WORD   SRC2.DSCR.PTR   ; ptr to src2 descriptor
        .WORD   '               ; extend with spaces
        BLO     LESS            ; src1<src2
        BEQ     EQUAL           ; src1=src2
        BHI     GREATER         ; src1>src2
```

3. Compare as far as the length of shorter of two strings - Register Form

```
        MOV     SRC1.DSCR,R0    ; 1st source descriptor
        MOV     SRC1.DSCR+2,R1
        MOV     SRC2.DSCR,R2    ; 2nd source descriptor
        MOV     SRC2.DSCR+2,R3
```

```
    CMP     R0,R2           ; length of shorter
    BHI     1$
    MOV     R0,R2
1$: MOV     R2,R0
                            ; no fill is used
    CMPC                    ; compare strings
    BEQ     EQUAL           ; use unsigned branches
    BNE     NOTEQL
```

Notes:

1. The operation of this instruction is unaffected by any overlap of the source character strings.

2. If the srcl character string is vacant, the fill character will be compared with src2. If the src2 character string is vacant, the fill character will be compared with srcl. If both character strings are vacant, the condition codes will indicate equality.

3. CMPC -- If an initial source character string descriptor is vacant, the resulting sub-string descriptor is the same as the original character string descriptor.

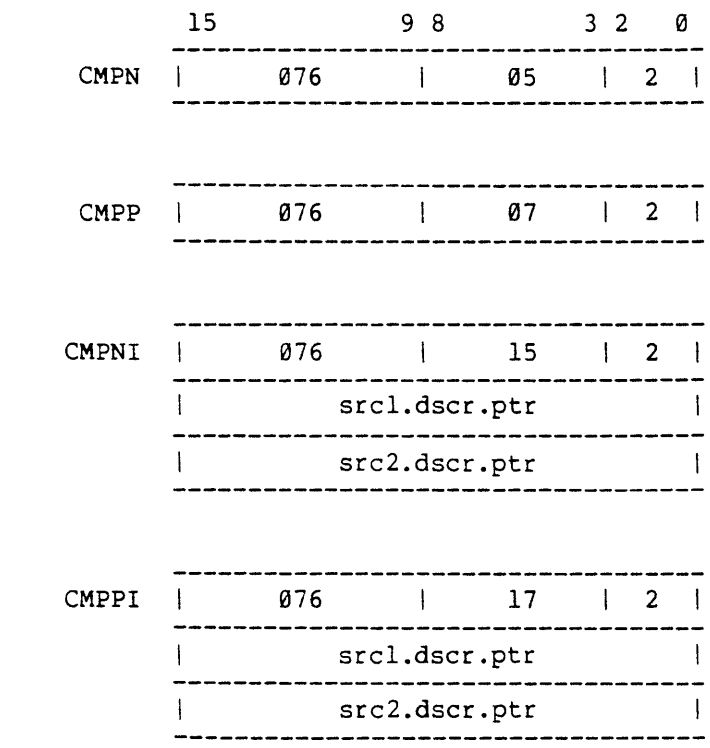
4. A test for success is BEQ; a test for failure is BNE.

5. When the instruction terminates, the condition codes will be set as if a CMPB instruction operated on the most significant unequal characters. If both strings are initially vacant or are identical, the condition codes will be set as if the last characters to be compared were identical. This results in equality with N cleared, Z set, V cleared, and C cleared.

6. Both CMPC and CMPCI update the condition codes. CMPC returns sub-string descriptors.

5.4 CMPN / CMPP / CMPNI / CMPPI - Compare Decimal

Format:



Operation:

Src1 is compared with src2 (src1-src2).

Condition Codes:

N: set if src1<src2; cleared otherwise
Z: set if src1=src2; cleared otherwise
V: cleared
C: cleared

Suspendability:

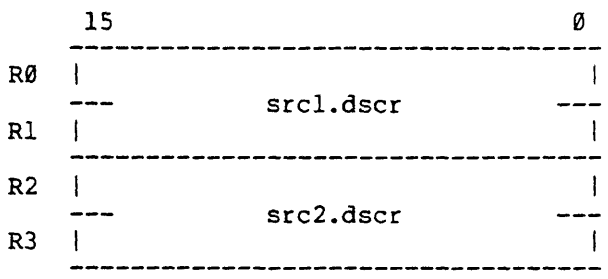
This instruction is potentially suspendable.

Description:

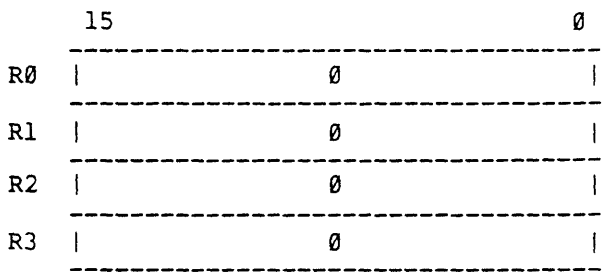
Src1 is arithmetically compared with src2. The condition codes reflect the comparison. The signed branch instruction can be used to test the result.

Register Form - CMPN and CMPP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, and the second source descriptor is placed in R2-R3:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form - CMPNI and CMPPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Compare Decimal Strings - Register Form

```
MOV     SRC1.DSCR,R0    ; 1st source descriptor
MOV     SRC1.DSCR+2,R1
MOV     SRC2.DSCR,R2    ; 2nd source descriptor
MOV     SRC2.DSCR+2,R3
```

```
CMPN / CMPP             ; compare
BLT     LESS            ; use signed branches
BEQ     EQUAL
BGT     GREATER
```

2. Compare Decimal Strings - In-line Form

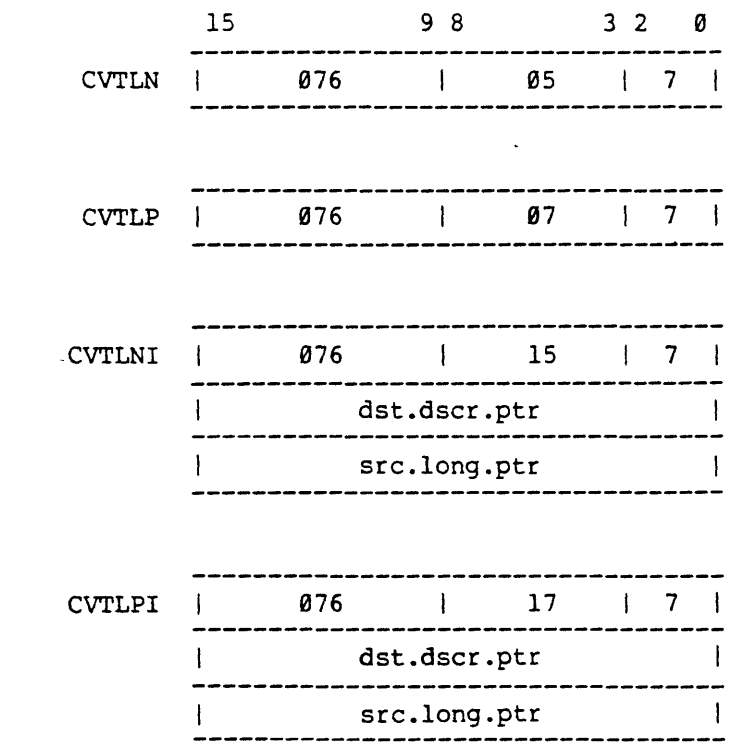
```
CMPNI / CMPPI           ; compare
.WORD   SRC1.DSCR.PTR   ; ptr to src1 descriptor
.WORD   SRC2.DSCR.PTR   ; ptr to src2 descriptor
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

5.5 CVTLN / CVTLP / CVTLNI / CVTLPI - Convert Long to Decimal

Format:



Operation:

decimal string <- long integer

Condition Codes:

N: set if dst<0; cleared otherwise
Z: set if dst=0; cleared otherwise
V: set if dst can not contain all significant digits of the result; cleared otherwise
C: cleared

Suspendability:

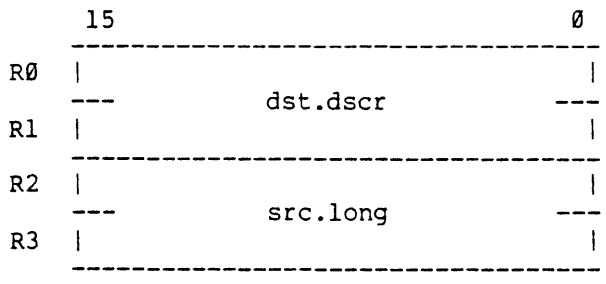
This instruction is potentially suspendable.

Description:

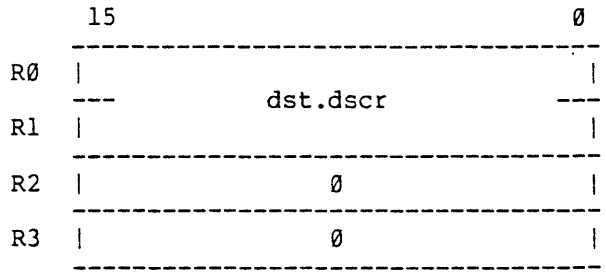
The source long integer is converted to a decimal string. The condition codes reflect the result stored in the destination decimal string, and whether all significant digits were stored.

Register Form - CVTLN and CVTLP

When the instruction starts, the operands must have been placed in the general registers. The destination descriptor is placed in R0-R1, and the source long integer is placed in R2-R3:



When the instruction is completed, the source long integer registers are cleared:



In-line Form - CVTLNI and CVTLPI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word decimal string destination descriptor, and a word address pointer to a two word long integer source. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Convert Long to Decimal - Register Form

```
MOV     DST.DSCR,RØ     ; destination descriptor
MOV     DST.DSCR+2,Rl
MOV     SRC.LONG+2,R2   ; source long integer
MOV     SRC.LONG,R3
CVTLN / CVTLP           ; convert
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

2. Convert Long to Decimal - In-line Form

```
CVTLNI / CVTLPI         ; convert
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
.WORD   SRC.LONG.PTR    ; ptr to long integer
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

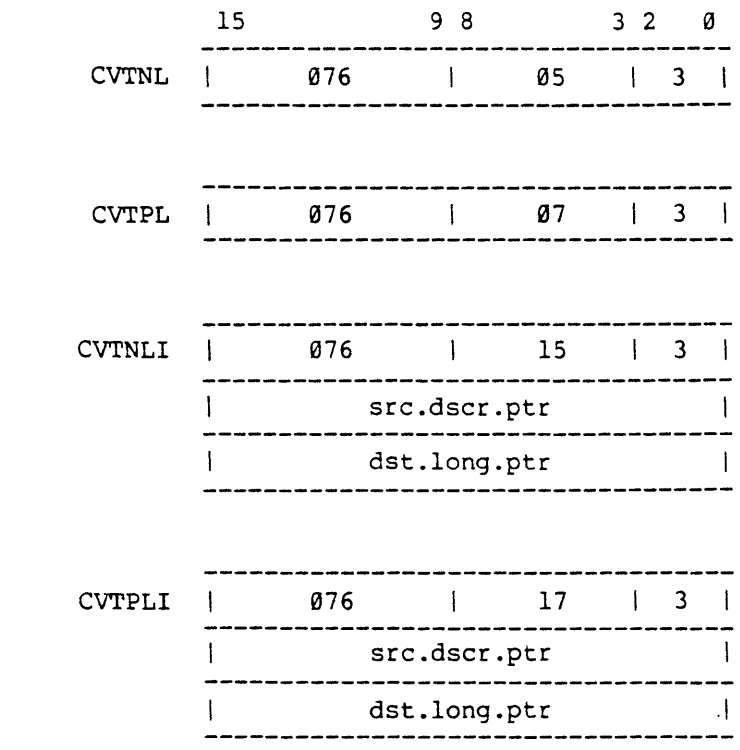
Notes:

1. Register forms use a long integer oriented with the sign and high order portion in R2, and the low order portion in R3.

2. In-line forms use a long integer oriented with the low order portion in src.long, and the sign and high order portion in src.long+2.

5.6 CVTNL / CVTPL / CVTNLI / CVTPLI - Decimal to Long

Format:



Operation:

long integer <- decimal string

Condition Codes:

The condition codes are based on the long integer destination and on the sign of the source decimal string.

N: set if long.integer<0; cleared otherwise
Z: set if long.integer=0; cleared otherwise
V: set if long.integer dst can not correctly represent the two's complement form of the result; cleared otherwise
C: set if src<0 and long.integer#0; cleared otherwise

Suspendability:

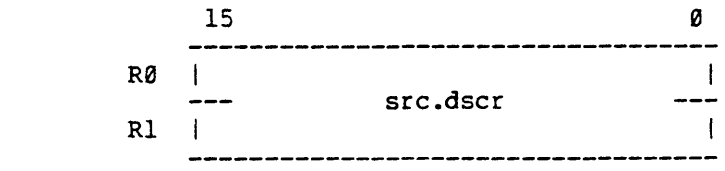
This instruction is potentially suspendable.

Description:

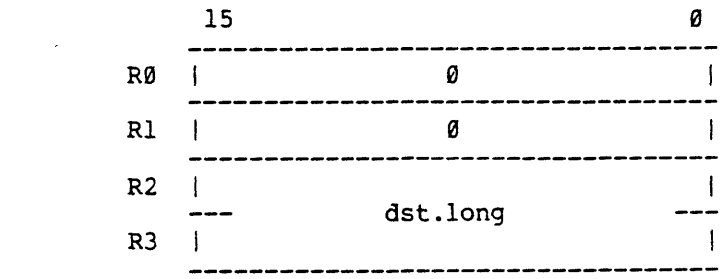
The source decimal string is converted to a long integer. The condition codes reflect the result of the operation, or whether significant digits were not converted.

Register Form - CVTNL and CVTPL

When the instruction starts, the operands must have been placed in the general registers. The source decimal string descriptor is placed in R0-R1:



When the instruction is completed, the source decimal string descriptor registers are cleared, and the destination long integer is returned in R2-R3:



In-line Form - CVTNLI and CVTPLI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word decimal string source descriptor, and a word address pointer to a two word long integer destination. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Convert Decimal to Long - Register Form

```
MOV      SRC.DSCR,RØ      ; source descriptor
MOV      SRC.DSCR+2,Rl
CVTNL / CVTPL             ; convert
BVS      OVERFLOW         ; check for error
BLT      NEGATIVE         ; negative destination
BEQ      EQUAL            ; zero destination
BGT      GREATER          ; positive destination
```

2. Convert Decimal to Long - In-line Form

```
CVTNLI / CVTPLI           ; convert
.WORD    SRC.DSCR.PTR     ; ptr to src descriptor
.WORD    DST.LONG.PTR     ; ptr to dst long int
BVS      OVERFLOW         ; check for error
BLT      NEGATIVE         ; negative destination
BEQ      EQUAL            ; zero destination
BGT      GREATER          ; positive destination
```

Notes:

1. Register forms use a long integer oriented with the sign and high order portion in R2, and the low order portion in R3.

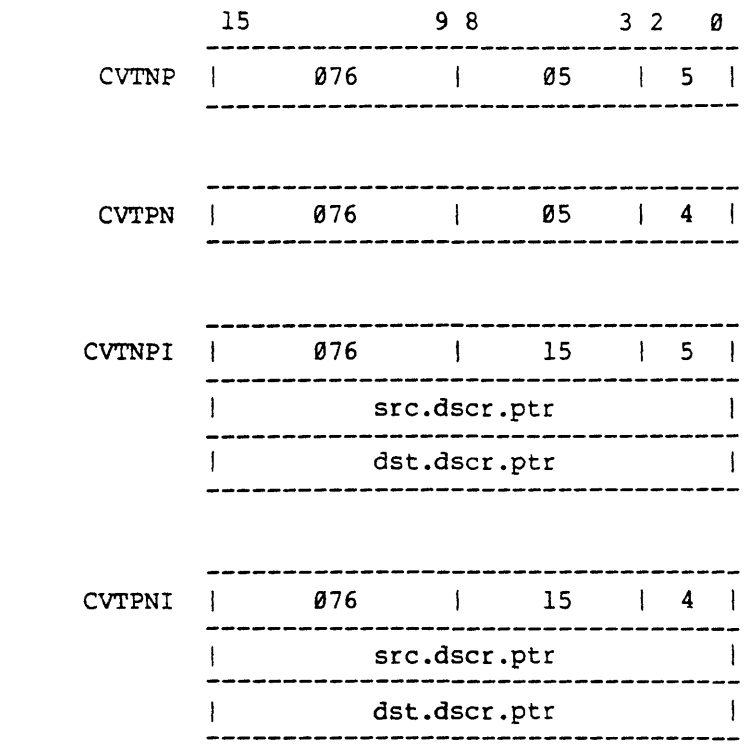
2. In-line forms use a long integer oriented with the low order portion in dst.long, and the sign and high order portion in dst.long+2.

3. If the V bit is set, the contents of the long integer destination are the least significant 32 bits of the result.

4. A source whose value is +2**31 can be represented as a 32 bit binary integer. However, since the destination is a two's complement long integer, the resulting condition codes will be N set, Z cleared, V set, and C cleared.

5.7 CVTNP / CVTPN / CVTNPI / CVTPNI - Convert Decimal

Format:



Operation:

```
CVTNP / CVTNPI    packed string <- numeric string
CVTPN / CVTPNI    numeric string <- packed string
```

Condition Codes:

```
N:  set if dst<0;  cleared otherwise
Z:  set if dst=0;  cleared otherwise
V:  set if dst can not contain all significant digits of the
    result;  cleared otherwise
C:  cleared
```

Suspendability:

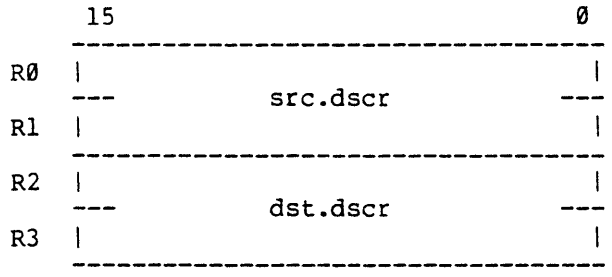
This instruction is potentially suspendable.

Description:

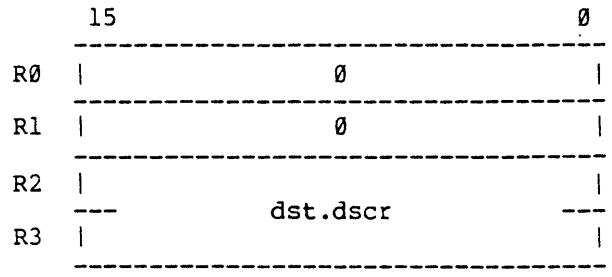
These instructions convert between numeric and packed decimal strings. The source decimal string is converted and moved to the destination string. The condition codes reflect the result of the operation, or whether all significant digits were stored.

Register Form - CVTNP and CVTPN

When the instruction starts, the operands must have been placed in the general registers. The source descriptor is placed in RØ-Rl, and the destination descriptor is placed in R2-R3:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form - CVTNPI and CVTPNI

Each word address pointer which follows the opcode word in the instruction stream refers to a two word decimal string descriptor. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Convert Between Numeric String and Packed String - Register Form

```
MOV      SRC.DSCR,RØ      ; source descriptor
MOV      SRC.DSCR+2,R1
MOV      DST.DSCR,R2      ; destination descriptor
MOV      DST.DSCR+2,R3
CVTNP / CVTPN             ; convert
BVS      OVERFLOW         ; check for error
BLT      NEGATIVE         ; negative destination
BEQ      EQUAL            ; zero destination
BGT      GREATER          ; positive destination
```

2. Convert Between Numeric String and Packed String - In-line Form

```
CVTNPI / CVTPNI           ; convert
.WORD    SRC.DSCR.PTR     ; ptr to src descriptor
.WORD    DST.DSCR.PTR     ; ptr to dst descriptor
BVS      OVERFLOW         ; check for error
BLT      NEGATIVE         ; negative destination
BEQ      EQUAL            ; zero destination
BGT      GREATER          ; positive destination
```

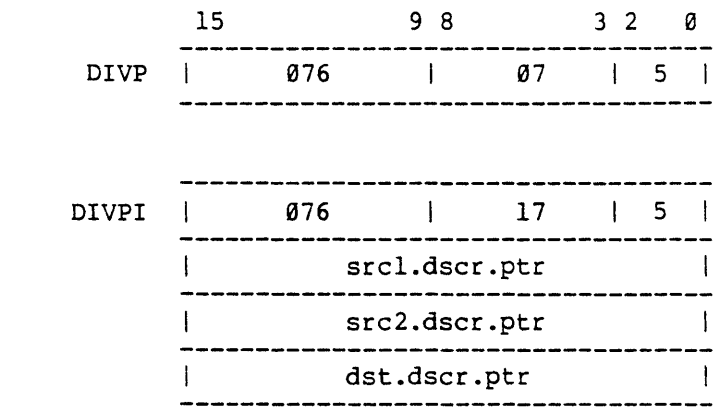
Notes:

1. The results of the instruction are unpredictable if the source and destination strings overlap.

2. These instructions use both a numeric and a packed decimal string descriptor.

5.8 DIVP / DIVPI - Divide Decimal

Format:



Operation:

dst <- src2 / src1

Condition Codes:

N: set if dst<0; cleared otherwise
Z: set if dst=0; cleared otherwise
V: set if dst can not contain all significant digits of the result or if src1=0; cleared otherwise
C: set if src1=0; cleared otherwise

Suspendability:

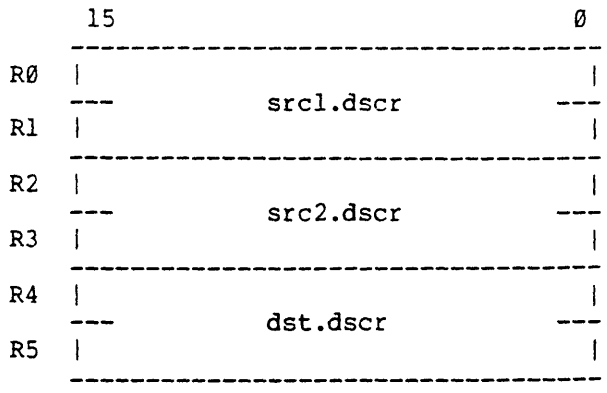
This instruction is potentially suspendable.

Description:

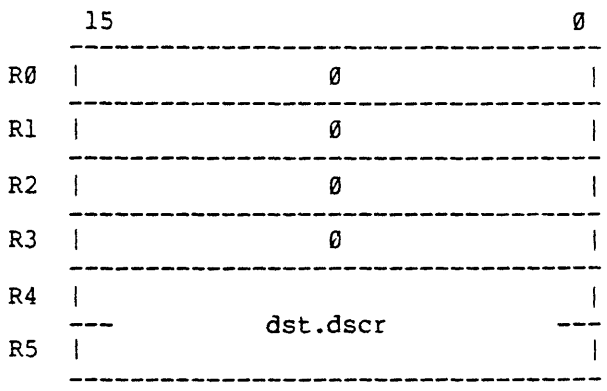
Src2 is divided by src1, and the quotient (fraction truncated) is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - DIVP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form - DIVPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two word decimal string descriptor. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Divide - Register Form

```
MOV     SRC1.DSCR,RØ    ; divisor descriptor
MOV     SRC1.DSCR+2,Rl
MOV     SRC2.DSCR,R2    ; dividend descriptor
MOV     SRC2.DSCR+2,R3
```

```
MOV     DST.DSCR,R4     ; quotient descriptor
MOV     DST.DSCR+2,R5
DIVP                    ; divide
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

2. Divide - In-line Form

```
DIVPI                   ; divide
.WORD   SRC1.DSCR.PTR   ; ptr to divisor dscr
.WORD   SRC2.DSCR.PTR   ; ptr to dividend dscr
.WORD   DST.DSCR.PTR    ; ptr to quotient dscr
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

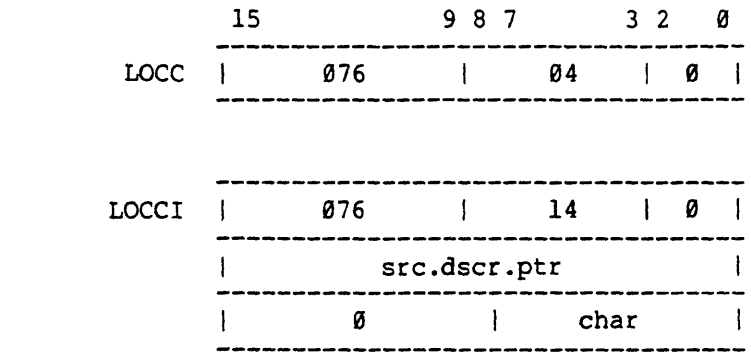
2. The results of the instruction are unpredictable if the source and destination strings overlap.

3. Division by zero will set the V and C bits. The destination string, and the N and Z condition code bits will be unpredictable.

4. No numeric string divide instruction is provided.

5.9 LOCC / LOCCI - Locate Character

Format:



Operation:

Search source character string for a character.

Condition Codes:

The condition codes are based on the final contents of RØ.

N: set if RØ<15> set; cleared otherwise
Z: set if RØ=Ø; cleared otherwise
V: cleared
C: cleared

Suspendability:

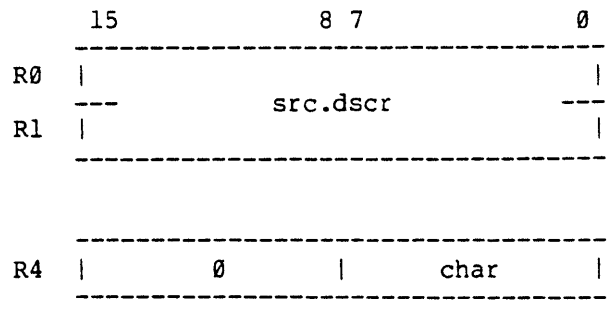
This instruction is potentially suspendable.

Description:

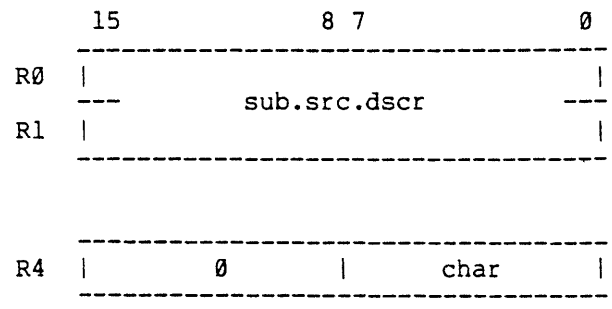
The source character string is searched from most significant to least significant character until the first occurrence of the search character. A character string descriptor is returned in RØ-R1 which represents the portion of the source character string beginning with the located character. If the source character string contains only characters not equal to the search character, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in RØ.

Register Form - LOCC
-------- ---- - ----

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the search character is placed in R4<7:0>, and R4<15:8> must be zero:

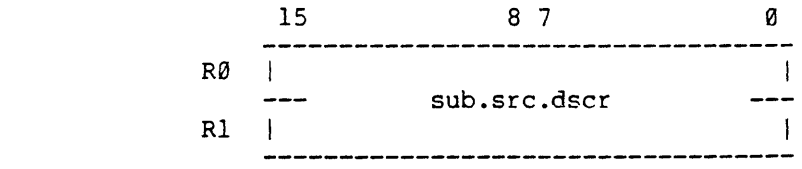


When the instruction is completed, R0-R1 contain a character set descriptor which represents the sub-string of the source character string beginning with the located character:



In-line Form - LOCCI
------- ---- - -----

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, and a word whose low order half contains the search character and whose high order half must be zero. When the instruction is completed, R0-R1 contain a character string descriptor which represents the sub-string of the source character string beginning with the located character. R2-R6 are unchanged:



Formal Description:

```
src.len = R0;          ! LOCC only
src.adr = R1;          !    .
char = R4<7:0>;        !    .

temp = M[R7];          ! LOCCI only
src.len = M[temp];     !    .
src.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
char = M[R7]<7:0>;     !    .
R7 = R7+2;             !    .

found = 0;
while (src.len nequ 0) and (found eqlu 0) do
         if M[src.adr] nequ char then
             begin
             src.len = src.len-1;
             src.adr = src.adr+1
             end
         else found = 1;

R0 = src.len;
R1 = src.adr;
R4 = 0<15:8>@char;     ! LOCC only

N = R0<15>;
Z = R0 eqlu 0;
V = 0;
C = 0;
```

Examples:

1. Find the Beginning of a Comment - Register Form

```
        MOV     STR.DSCR,R0     ; string to search
        MOV     STR.DSCR+2,R1
        MOV     #';,R4          ; search for semi-colon
        LOCC                    ; locate
        BNE     FOUND           ; R0 and R1 are the
                                ; sub-string descriptor
```

2. Find the Beginning of a Comment - In-Line Form

```
LOCCI                       ; locate
.WORD   SRC.DSCR.PTR        ; ptr to src descriptor
.WORD   ';                  ; search for semi-colon
BNE     FOUND               ; R0 and R1 are the
                            ; sub-string descriptor
```

Notes:

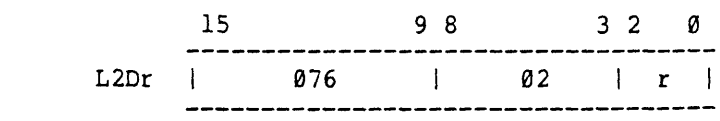
1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match was found. The original source character string descriptor is returned in R0-R1.

2. A test for success is BNE; a test for failure is BEQ.

3. The condition codes will be set as if this instruction were followed by TST R0.

5.10 L2Dr - Load 2 Descriptors

Format:



Operation:

Load word pairs into RØ-Rl and R2-R3.

Condition Codes:

The condition codes are not affected.

N: not affected
Z: not affected
V: not affected
C: not affected

Suspendability:

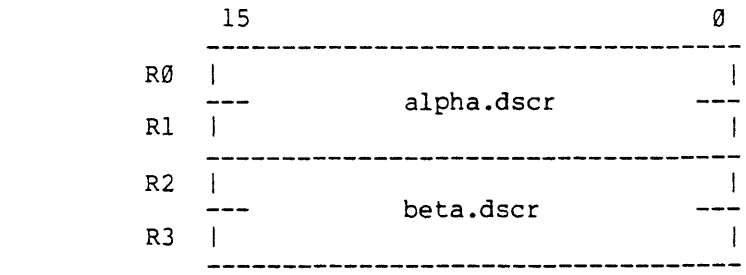
This instruction is non-suspendable.

Description:

This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers.

A descriptor 'alpha' is loaded into RØ-Rl; a second descriptor 'beta' is loaded into R2-R3. The address of the descriptors are determined by the addressing mode @(Rr)+ where r is the low order three bits of the opcode word. The address of the descriptor 'alpha' is derived by applying this addressing mode once; the address of the descriptor 'beta' is derived by applying this addressing mode a second time. The addressing mode auto-increments the indicated register by 2. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the 'alpha' descriptor is in R0-R1 and the 'beta' descriptor is in R2-R3:



Formal Description:

```
temp = R[r];
adr.alpha = M[temp]; temp = temp+2;
adr.beta = M[temp]; temp = temp+2;
if (r gequ 4) then R[r] = temp;
R0 = M[adr.alpha];
R1 = M[adr.alpha+2];
R2 = M[adr.beta];
R3 = M[adr.beta+2];
```

Examples:

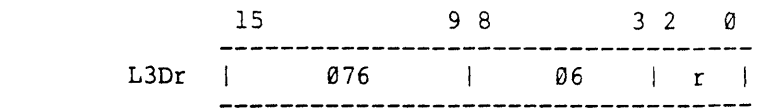
1. Decimal String Compare

```
        L2D7                    ; load descriptors
        .WORD   SRC1
        .WORD   SRC2
        CMPN                    ; compare
          .
          .
          .
SRC1:.WORD  SRC1.LEN            ; 1st src descriptor
    .WORD   SRC1.ADR
          .
          .
          .
SRC2:.WORD  SRC2.LEN            ; 2nd src descriptor
    .WORD   SRC2.ADR
```

Notes:

5.11 L3Dr - Load 3 Descriptors

Format:



Operation:

Load word pairs into RØ-Rl, R2-R3 and R4-R5.

Condition Codes:

The condition codes are not affected.

N: not affected
Z: not affected
V: not affected
C: not affected

Suspendability:

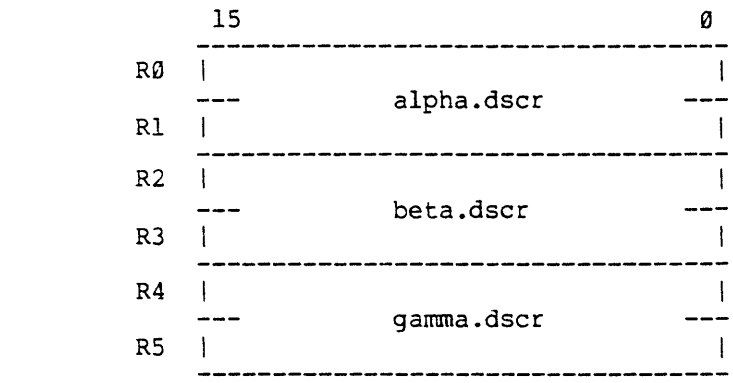
This instruction is non-suspendable.

Description:

This instruction augments the character and decimal string instructions by efficiently loading string descriptors into the general registers.

A descriptor 'alpha' is loaded into RØ-Rl; a second descriptor 'beta' is loaded into R2-R3; a third descriptor 'gamma' is loaded into R4-R5. The address of the descriptors are determined by the addressing mode @(Rr)+ where r is the low order three bits of the opcode word. The address of the descriptor 'alpha' is derived by applying this addressing mode once; the address of the descriptor 'beta' is derived by applying this addressing mode a second time; the address of the descriptor 'gamma' is derived by applying this addressing mode a third time. The address mode auto-increments the indicated register by 2. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

When the instruction is completed, the 'alpha' descriptor is in R0-R1, the 'beta' descriptor is in R2-R3 and the 'gamma' descriptor is in R4-R5:



Formal Description:

```
temp = R[r];
adr.alpha = M[temp]; temp = temp+2;
adr.beta = M[temp]; temp = temp+2;
adr.gamma = M[temp]; temp = temp+2;
if (r gequ 6) then R[r] = temp;
R0 = M[adr.alpha];
R1 = M[adr.alpha+2];
R2 = M[adr.beta];
R3 = M[adr.beta+2];
R4 = M[adr.gamma];
R5 = M[adr.gamma+2];
```

Examples:

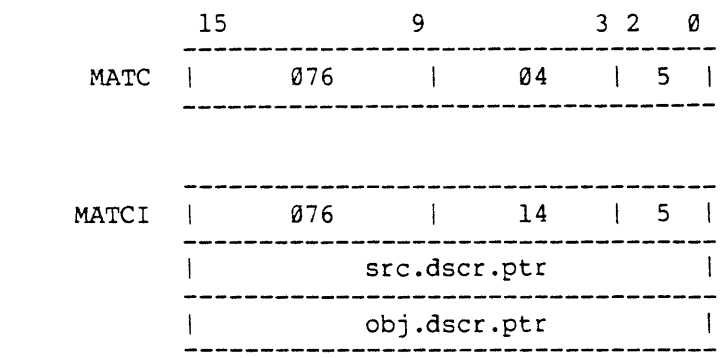
1. Three Address Add

```
        L3D7                        ; load descriptors
        .WORD   SRC1
        .WORD   SRC2
        .WORD   DST
        ADDN                        ; add
          .
          .
          .
SRC1:.WORD  SRC1.LEN                ; 1st src descriptor
        .WORD   SRC1.ADR
          .
          .
          .
SRC2:.WORD  SRC2.LEN                ; 2nd src descriptor
        .WORD   SRC2.ADR
          .
          .
          .
DST:.WORD   DST.LEN                 ; dst descriptor
        .WORD   DST.ADR
```

Notes:

5.12 MATC / MATCI - Match Character

Format:



Operation:

Search source character string for object character string.

Condition Codes:

The condition codes are based on the final contents of R0.

N: set if R0<15> set; cleared otherwise
Z: set if R0=0; cleared otherwise
V: cleared
C: cleared

Suspendability:

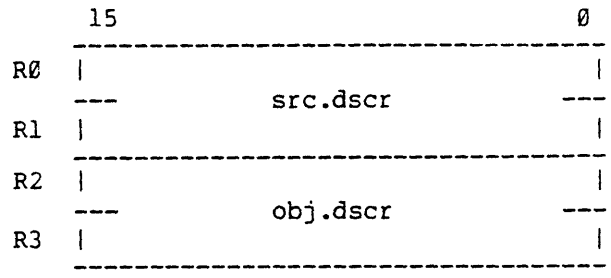
This instruction is potentially suspendable.

Description:

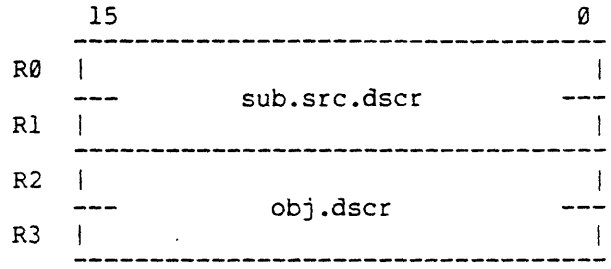
The source character string is searched from most significant to least significant character for the first occurrence of the entire object character string. A character string descriptor is returned in R0-R1 which represents the portion of the original source character string beginning with the most significant character to completely match the object character string. If the object character string did not completely match any portion of the source character string, the character descriptor returned in R0-R1 is vacant with an address one greater than the least significant character in the source string. The condition codes reflect the resulting value in R0. If the Z bit is cleared, the entire object was successfully matched with the source character string; if the Z bit is set, the match failed.

Register Form - MATC
-------- ---- - ----

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in RØ-Rl, and the object character string descriptor is placed in R2-R3:

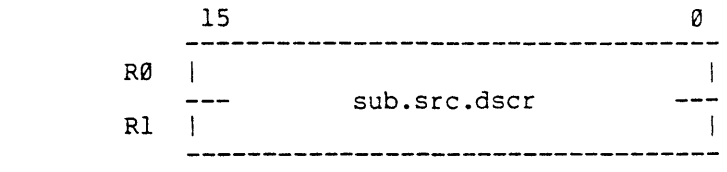


The instruction terminates with a character sub-string descriptor returned in RØ-Rl which represents the portion of the original source character string beginning with the most significant character to completely match the object character string.



In-line Form - MATCI
------- ---- - -----

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, and a word address pointer to a two word character string object descriptor. The instruction terminates with a character sub-string descriptor returned in RØ-Rl which represents the portion of the original source character string beginning with the most significant character to completely match the object character string. R2-R6 are unchanged when the instruction is completed.



Formal Description:

```
    src.len = R0;          ! MATC only
    src.adr = R1;          !    .
    obj.len = R2;          !    .
    obj.adr = R3;          !    .

    temp = M[R7];          ! MATCI only
    src.len = M[temp];     !    .
    src.adr = M[temp+2];   !    .
    R7 = R7+2;             !    .
    temp = M[R7];          !    .
    obj.len = M[temp];     !    .
    obj.adr = M[temp+2];   !    .
    R7 = R7+2;             !    .

    tmp.len = obj.len;
    found = 0;
    while (src.len gequ obj.len) and (obj.len nequ 0)
      and (found eqlu 0) do
            begin
            same = 1;
            while (obj.len nequ 0) and (same eqlu 1) do
                if (M[obj.adr] eqlu M[src.adr])
                then
                    begin
                    obj.len = obj.len-1;
                    obj.adr = obj.adr+1;
                    src.len = src.len-1;
                    src.adr = src.adr+1
                    end
                else
                    same = 0;
            found = same;
            obj.adr = obj.adr+obj.len-tmp.len;
            src.len = src.len+tmp.len-obj.len-1;
            src.adr = src.adr+obj.len-tmp.len+1;
            obj.len = tmp.len
            end;
    if found eql 1
    then
            begin
            R0 = src.len+1;
            R1 = src.adr-1
            end
```

```
else
        begin
        R0 = 0;
        R1 = src.adr+src.len
        end;

R2 = obj.len;          ! MATC only
R3 = obj.adr;          !

N = R0<15>;
Z = R0 eqlu 0;
V = 0;
C = 0;
```

Examples:

1. Find a Keyword - Register Form

```
MOV     SRC.DSCR,R0     ; 1st source descriptor
MOV     SRC.DSCR+2,R1
MOV     OBJ.DSCR,R2     ; 2nd source descriptor
MOV     OBJ.DSCR+2,R3
MATC                    ; search for keyword
BNE     FOUND           ; object was in string
```

2. Find a Keyword - In-line Form

```
MATCI                   ; search for keyword
.WORD   SRC.DSCR.PTR    ; ptr to src descriptor
.WORD   OBJ.DSCR.PTR    ; ptr to obj descriptor
BNE     FOUND           ; object was in string
```

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and object character strings.

2. A vacant object character string matches any non-vacant source character string. A vacant source character string will not match any object character string. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match was found. The original source character string descriptor is returned in R0-R1.

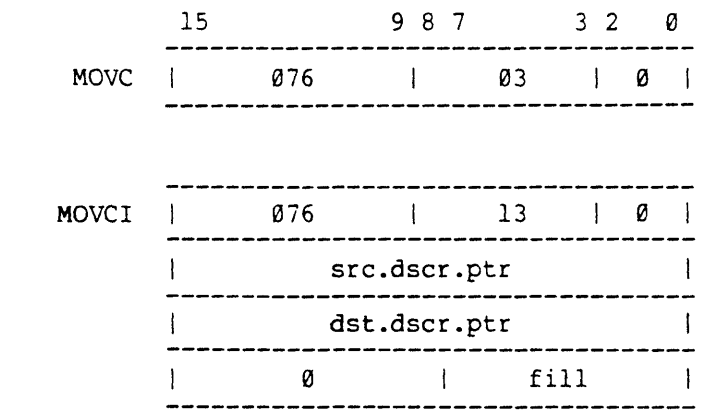
3. If the length of the object character string is greater than that of the source character string then no match is found; R0-R1 and the condition codes will be updated.

4. A test for success is BNE; a test for failure is BEQ.

5. The condition codes will be set as if this instruction were followed by TST RØ.

5.16 MOVC / MOVCI - Move Character

Format:



Operation:

dst <- src

Condition Codes:

The condition codes are based on the arithmetic comparison of the initial character string lengths (result=src.len-dst.len).

N: set if result<0; cleared otherwise
Z: set if result=0; cleared otherwise
V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len-dst.len); cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Suspendability:

This instruction is potentially suspendable.

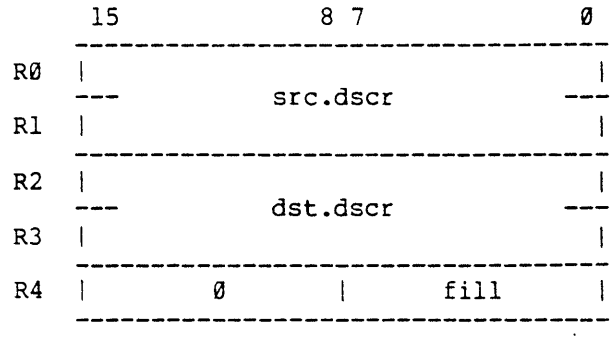
Description:

The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the most significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set.

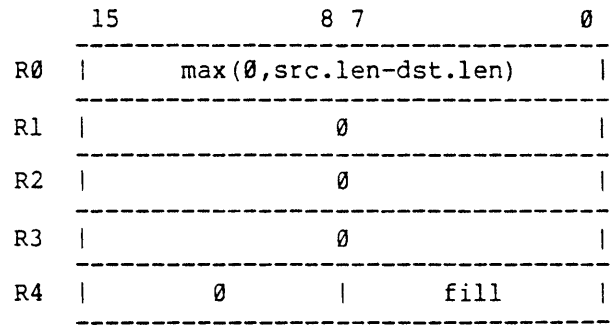
If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form - MOVC
-------- ---- - ----

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero:



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared:



In-line Form - MOVCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, a word address pointer to a two word character string destination descriptor, and a word whose low order half contains the fill character and whose high order half must be zero. R0-R6 are unchanged when the instruction is completed.

Formal Description:

```
src.len = R0;          ! MOVC only
src.adr = R1;          !    .
dst.len = R2;          !    .
dst.adr = R3;          !    .
fill = R4<7:0>;        !    .

temp = M[R7];          ! MOVCI only
src.len = M[temp];     !    .
src.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
temp = M[R7];          !    .
dst.len = M[temp];     !    .
dst.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
fill = M[R7]<7:0>;     !    .
R7 = R7+2;             !    .

carry@temp = src.len-dst.len;
N = temp<15>;
Z = temp eqlu 0;
V = (src.len<15> neq dst.len<15>) and (src.len<15> eql
    temp<15>)
C = carry;

if src.adr gequ dst.adr then
         begin       ! most to least significant
 characters
         while (src.len nequ 0) and (dst.len nequ 0) do
             begin
             M[dst.adr] = M[src.adr];
             src.len = src.len-1;
             src.adr = src.adr+1;
             dst.len = dst.len-1;
             dst.adr = dst.adr+1
             end;
         while dst.len nequ 0 do
             begin
             M[dst.adr] = fill;
             dst.len = dst.len-1;
             dst.adr = dst.adr+1
```

```
            end
        end
else
        begin       ! least to most significant
 characters
        src.adr = src.len-1-max(0,src.len-dst.len)+src.adr;
        dst.adr = dst.len+dst.adr-1;
        while src.len lssu dst.len do
            begin
            M[dst.adr] = fill;
            dst.len = dst.len-1;
            dst.adr = dst.adr-1
            end;
        while dst.len nequ 0 do
            begin
            M[dst.adr] = M[src.adr];
            src.len = src.len-1;
            src.adr = src.adr-1;
            dst.len = dst.len-1;
            dst.adr = dst.adr-1
            end
        end;

R0 = src.len;         ! MOVC only
R1 = 0;               !    .
R2 = 0;               !    .
R3 = 0;               !    .
R4 = 0<15:8>@fill;    !    .
```

Examples:

1. Moving Data - Register Form

```
        MOV     SRC.DSCR,R0     ; source descriptor
        MOV     SRC.DSCR+2,R1
        MOV     DST.DSCR,R2     ; destination descriptor
        MOV     DST.DSCR+2,R3
        MOV     #' ,R4          ; fill with spaces
        MOVC                    ; move
        BHI     TRUNC           ; test for truncation
        BLO     FILL            ; test for fill
        BEQ     EQUAL           ; test for equal length
```

2. Moving Data - In-line Form

```
        MOVCI                   ; move
        .WORD   SRC.DSCR.PTR    ; ptr to src descriptor
        .WORD   DST.DSCR.PTR    ; ptr to dst descriptor
        .WORD   '               ; fill is space
        BHI     TRUNC           ; test for truncation
        BLO     FILL            ; test for fill
        BEQ     EQUAL           ; test for equal length
```

3. Clearing Storage - Register Form

```
CLR     RØ              ; zero length source
MOV     DST.DSCR,R2     ; destination descriptor
MOV     DST.DSCR+2,R3
CLR     R4              ; store null characters
MOVC                    ; propagate fill
```

4. Clearing Storage - In-line Form

```
MOVCI                   ; propagate fill
.WORD   SRC.DSCR.PTR    ; ptr to null str dscr
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
.WORD   Ø               ; fill with nulls
```

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.

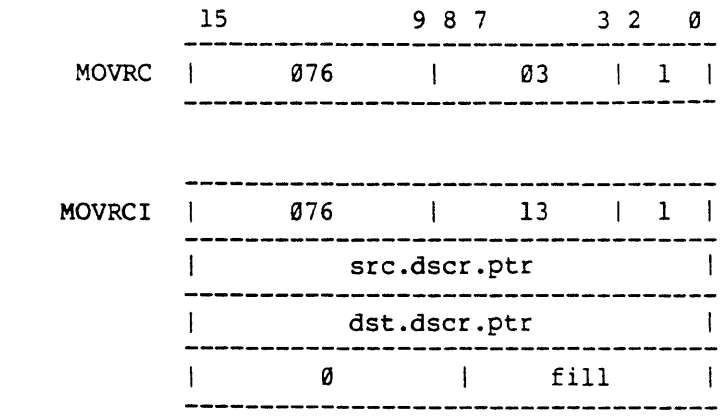
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. The condition codes will be updated. MOVC will update the general registers.

3. MOVC -- When the instruction terminates, RØ is zero only if Z or C are set.

4. The condition codes will be set as if this instruction were preceded by CMP src.len,dst.len.

5.17 MOVRC / MOVRCI - Move Reverse Justified Character

Format:



Operation:

dst <- reverse justified src

Condition Codes:

The condition codes are based on the arithmetic comparison of the initial character string lengths (result=src.len-dst.len).

N: set if result<0; cleared otherwise
Z: set if result=0; cleared otherwise
V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len-dst.len); cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Suspendability:

This instruction is potentially suspendable.

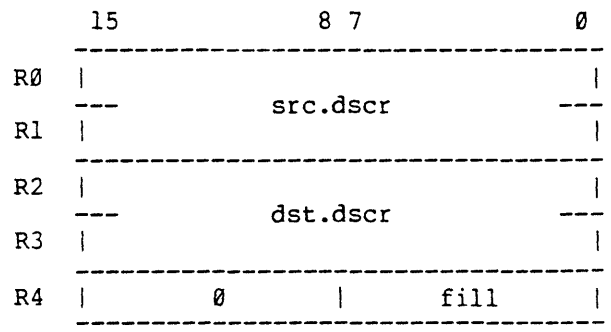
Description:

The character string specified by the source descriptor is moved into the area specified by the destination descriptor. It is aligned by the least significant character. The condition codes reflect an arithmetic comparison of the original source and destination lengths. If the source string is shorter than the destination string, the fill character is used to complete the most significant part of the destination string. This is indicated by the C bit set.

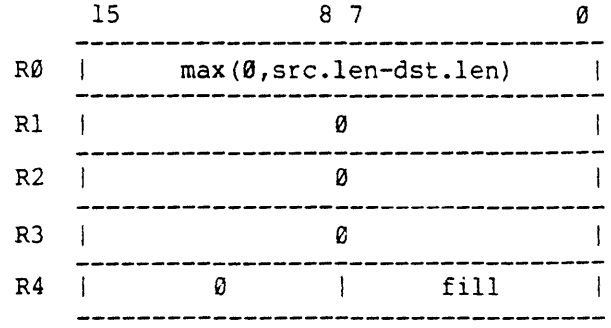
If the source string is longer than the destination string, the most significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form - MOVRC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in RØ-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:Ø>, and R4<15:8> must be zero:



When the instruction is completed, RØ contains the number of unmoved source string characters, and R1 through R3 are cleared:



In-line Form - MOVRCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, a word address pointer to a two word character string destination descriptor, and a word whose low order half contains the fill character and whose high order half must be zero. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

```
src.len = RØ;          ! MOVRC only
src.adr = Rl;          !    .
dst.len = R2;          !    .
dst.adr = R3;          !    .
fill = R4<7:Ø>;        !    .

temp = M[R7];          ! MOVRCI only
src.len = M[temp];     !    .
src.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
temp = M[R7];          !    .
dst.len = M[temp];     !    .
dst.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
fill = M[R7]<7:Ø>;     !    .
R7 = R7+2;             !    .

carry@temp = src.len-dst.len;
N = temp<15>;
Z = temp eqlu Ø;
V = (src.len<15> neq dst.len<15>) and (src.len<15> eql temp<15>)
C = carry;

if (src.len+src.adr-1) gequ (dst.len+dst.adr-1) then
    begin              ! most to least significant
 characters
        src.adr = max(Ø,src.len-dst.len)+src.adr;
        while src.len lssu dst.len do
            begin
            M[dst.adr] = fill;
            dst.len = dst.len-1;
            dst.adr = dst.adr+1
            end;
        while dst.len nequ Ø do
            begin
            M[dst.adr] = M[src.adr];
            src.len = src.len-1;
            src.adr = src.adr+1;
            dst.len = dst.len-1;
            dst.adr = dst.adr+1
```

```
            end;
        end
else
        begin                ! least to most significant
 characters
        src.adr = src.len+src.adr-1;
        dst.adr = dst.len+dst.adr-1;
        while (src.len nequ Ø) and (dst.len nequ Ø) do
            begin
            M[dst.adr] = M[src.adr];
            src.len = src.len-1;
            src.adr = src.adr-1;
            dst.len = dst.len-1;
            dst.adr = dst.adr-1
            end;
        while dst.len nequ Ø do
            begin
            M[dst.adr] = fill;
            dst.len = dst.len-1;
            dst.adr = dst.adr-1
            end
        end;

R0 = src.len;          ! MOVRC only
R1 = 0;                !    .
R2 = 0;                !    .
R3 = 0;                !    .
R4 = 0<15:8>@fill;     !    .
```

Examples:

1. Moving Data - Register Form

```
        MOV     SRC.DSCR,R0      ; source descriptor
        MOV     SRC.DSCR+2,R1
        MOV     DST.DSCR,R2      ; destination descriptor
        MOV     DST.DSCR+2,R3
        MOV     #' ,R4           ; fill with spaces
        MOVRC                    ; move
        BHI     TRUNC            ; test for truncation
        BLO     FILL             ; test for fill
        BEQ     EQUAL            ; test for equal length
```

2. Moving Data - In-line Form

```
        MOVRCI                   ; move
        .WORD   SRC.DSCR.PTR     ; ptr to src descriptor
        .WORD   DST.DSCR.PTR     ; ptr to dst descriptor
        .WORD   '                ; fill is space
        BHI     TRUNC            ; test for truncation
        BLO     FILL             ; test for fill
        BEQ     EQUAL            ; test for equal length
```

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.

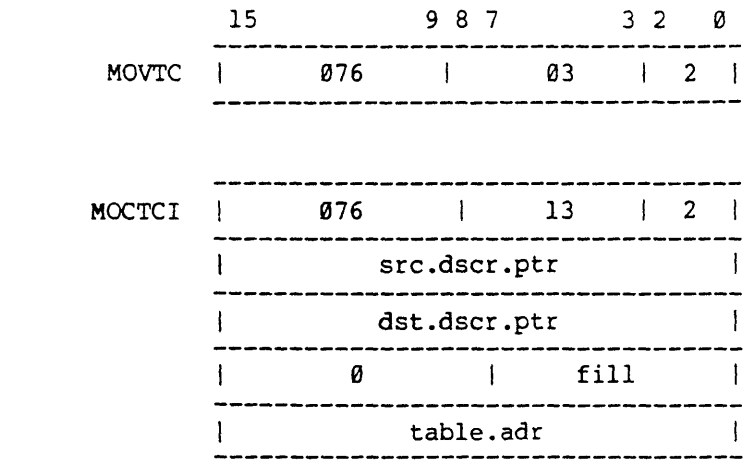
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVRC will update the general registers.

3. MOVRC -- When the instruction terminates, RØ is zero only if Z or C are set.

4. The condition codes will be set as if this instruction were preceded by CMP src.len,dst.len.

5.18 MOVTC / MOVTCI - Move Translated Character

Format:



Operation:

dst <- translated src

Condition Codes:

The condition codes are based on the arithmetic comparison of the initial character string lengths (result=src.len-dst.len).

N: set if result<Ø; cleared otherwise
Z: set if result=Ø; cleared otherwise
V: set if there was arithmetic overflow, that is, src.len<15> and dst.len<15> were different, and dst.len<15> was the same as bit <15> of (src.len-dst.len); cleared otherwise
C: cleared if there was a carry from the most significant bit of the result; set otherwise

Suspendability:

This instruction is potentially suspendable.

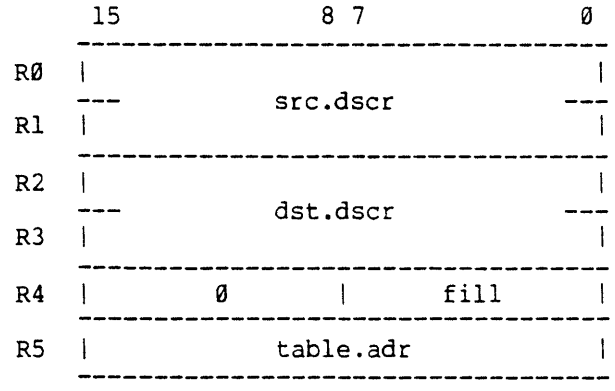
Description:

The character string specified by the source descriptor is translated and moved into the area specified by the destination descriptor. It is aligned by the most significant character. Translation is accomplished by using each source character as an 8 bit positive integer index into a 256 byte table, the address of which is an operand of the instruction. The byte at the indexed location in the table is stored in the destination string. The condition codes reflect an arithmetic comparison of the original contents source and destination lengths.

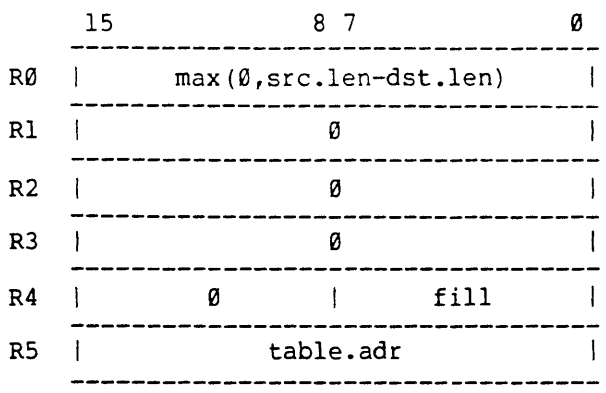
If the source string is shorter than the destination string, the untranslated fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are translated and moved with neither truncation nor filling. This is indicated by the Z bit set. The unsigned branch instructions may test the result of the instruction.

Register Form - MOVTC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, R4<15:8> must be zero, and the translation table address is placed in R5:



When the instruction is completed, R0 contains the number of unmoved source string characters, and R1 through R3 are cleared:



In-line Form - MOVTCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, a word address pointer to a two word character string destination descriptor, a word whose low order half contains the fill character and whose high order half must be zero, and a word containing the address of the translation table. R0-R6 are unchanged when the instruction is completed.

Formal Description:

```
src.len = R0;          ! MOVTC only
src.adr = R1;          !    .
dst.len = R2;          !    .
dst.adr = R3;          !    .
fill = R4<7:0>;        !    .
table.adr = R5;        !    .

temp = M[R7];          ! MOVTCI only
src.len = M[temp];     !    .
src.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
temp = M[R7];          !    .
dst.len = M[temp];     !    .
dst.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
fill = M[R7]<7:0>;     !    .
R7 = R7+2;             !    .
table.adr = M[R7];     !    .
R7 = R7+2;             !    .

carry@temp = src.len-dst.len;
N = temp<15>;
Z = temp eqlu 0;
```

```
V = (src.len<15> neq dst.len<15>) and (src.len<15> eql temp<15>)
C = carry;

if src.adr gequ dst.adr then
        begin               ! most to least significant
 characters
        while (src.len nequ Ø) and (dst.len nequ Ø) do
            begin
            M[dst.adr] = M[table.adr+M[src.adr]];
            src.len = src.len-1;
            src.adr = src.adr+1;
            dst.len = dst.len-1;
            dst.adr = dst.adr+1
            end;
        while dst.len nequ Ø do
            begin
            M[dst.adr] = fill;
            dst.len = dst.len-1;
            dst.adr = dst.adr+1
            end;
        end
else
        begin               ! least to most significant
 characters
        src.adr = src.len-1-max(Ø,src.len-dst.len)+src.adr;
        dst.adr = dst.len+dst.adr-1;
        while src.len lssu dst.len do
            begin
            M[dst.adr] = fill;
            dst.len = dst.len-1;
            dst.adr = dst.adr-1
            end;
        while dst.len nequ Ø do
            begin
            M[dst.adr] = M[table.adr+M[src.adr]];
            src.len = src.len-1;
            src.adr = src.adr-1;
            dst.len = dst.len-1;
            dst.adr = dst.adr-1
            end
        end;

RØ = src.len;        ! MOVTC only
R1 = Ø;              !    .
R2 = Ø;              !    .
R3 = Ø;              !    .
R4 = Ø<15:8>@fill;   !    .
R5 = table.adr;      !    .
```

Examples:

1. Character Code Conversion - Register Form

```
MOV     SRC.DSCR,RØ     ; EBCDIC source
MOV     SRC.DSCR+2,Rl
MOV     DST.DSCR,R2     ; ASCII destination
MOV     DST.DSCR+2,R3
MOV     #' ,R4          ; fill with ASCII spaces
MOV     #TABLE,R5       ; translation table
MOVTC                   ; translate and move
BHI     TRUNC           ; source was truncated
BLO     FILL            ; test for fill
BEQ     EQUAL           ; test for equal length
```

2. Character Code Conversion - In-line Form

```
MOVTCI                  ; translate and move
  -
.WORD   SRC.DSCR.PTR    ; ptr to src descriptor
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
.WORD   '               ; fill is space
BHI     TRUNC           ; test for truncation
BLO     FILL            ; test for fill
BEQ     EQUAL           ; test for equal length
```

Notes:

1. The operation of this instruction is unaffected by any overlap of the source and destination strings. The result is equivalent to having read the entire source string before storing characters in the destination.

2. If the destination string overlaps the translation table in any way, the results of the instruction will be unpredictable.

3. If the source string is vacant, the untranslated fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVTC will update the general registers.

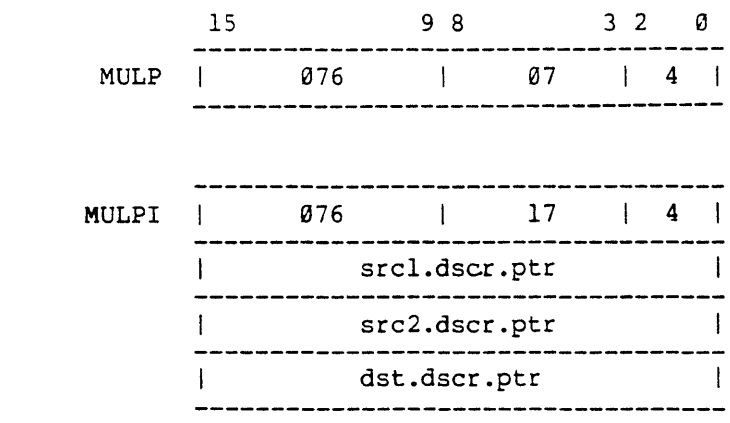
4. MOVTC -- When the instruction terminates, RØ is zero only if Z or C are set.

5. The condition codes will be set as if this instruction were preceded by CMP src.len,dst.len.

6. The effect of the instruction is unpredictable if the entire 256 byte translation table is not in readable memory.

5.19 MULP / MULPI - Multiply Decimal

Format:



Operation:

dst <- src2 * srcl

Condition Codes:

N: set if dst<Ø; cleared otherwise
Z: set if dst=Ø; cleared otherwise
V: set if dst can not contain all significant digits of the result; cleared otherwise
C: cleared

Suspendability:

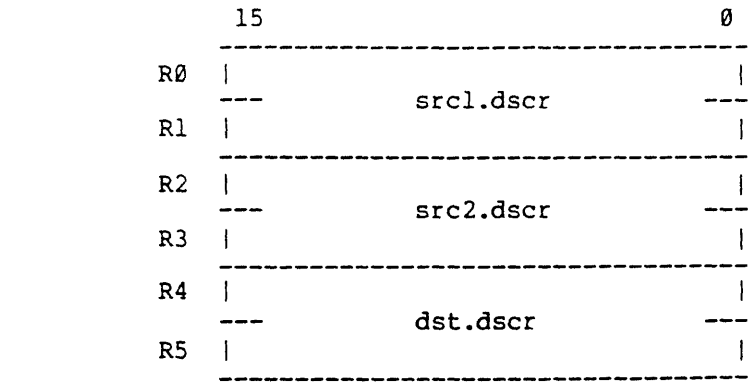
This instruction is potentially suspendable.

Description:

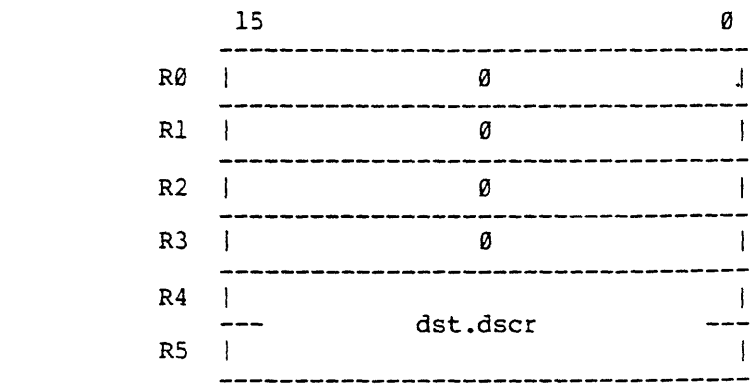
Srcl and src2 are multiplied, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - MULP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in RØ-Rl, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form - MULPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Multiply - Register Form

```
MOV     SRC1.DSCR,R0    ; 1st source descriptor
MOV     SRC1.DSCR+2,R1
MOV     SRC2.DSCR,R2    ; 2nd source descriptor
MOV     SRC2.DSCR+2,R3
```

```
MOV     DST.DSCR,R4     ; destination descriptor
MOV     DST.DSCR+2,R5
MULP                    ; multiply
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

2. Multiply - In-line Form

```
MULPI                   ; multiply
.WORD   SRC1.DSCR.PTR   ; ptr to src1 descriptor
.WORD   SRC2.DSCR.PTR   ; ptr to src2 descriptor
.WORD   DST.DSCR.PTR    ; ptr to dst descriptor
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

Notes:

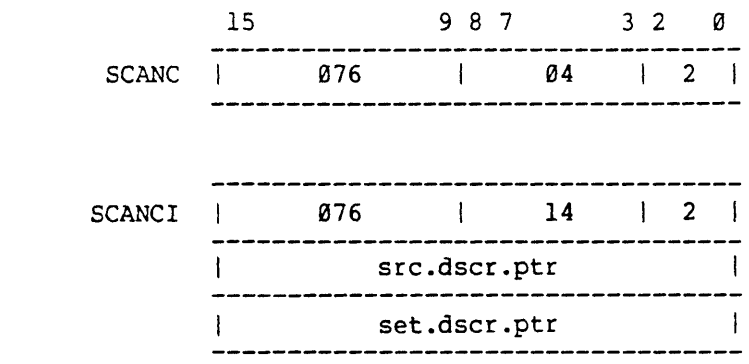
1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

2. The results of the instruction are unpredictable if the source and destination strings overlap.

3. No numeric string multiply instruction is provided.

5.20 SCANC / SCANCI - Scan Character

Format:



Operation:

Search source character string for a member of the character set.

Condition Codes:

The condition codes are based on the final contents of R0.

N: set if R0<15> set; cleared otherwise
Z: set if R0=0; cleared otherwise
V: cleared
C: cleared

Suspendability:

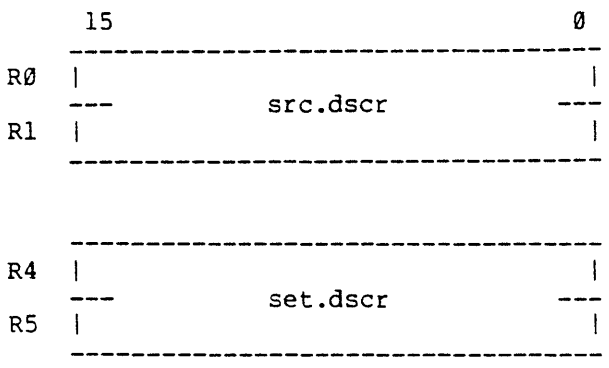
This instruction is potentially suspendable.

Description:

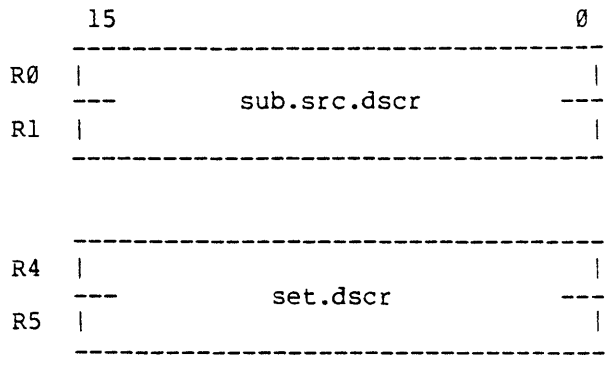
The source character string is searched from most significant to least significant character until the first occurrence of a character which is a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the located member of the character set. If the source character string contains only characters which are not in the character set, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form - SCANC
-------- ---- - -----

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in RØ-Rl, and the character set descriptor is placed in R4-R5:

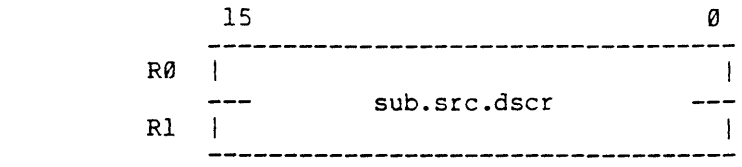


When the instruction is completed, RØ-Rl contain a character string descriptor which represents the sub-string of the source character string beginning with the character which is a member of the character set:



In-line Form - SCANCI
------- ---- - ------

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, and a word address pointer to a two word character set descriptor. When the instruction is completed, RØ-Rl contain a character string descriptor which represents the sub-string of the source character string beginning with the character which is a member of the character set. R2-R6 are unchanged:



Formal Description:

```
    src.len = R0;          ! SCANC only
    src.adr = R1;          !    .
    mask = R4<7:0>;        !    .
    table.adr = R5;        !    .

    temp = M[R7];          ! SCANCI only
    src.len = M[temp];     !    .
    src.adr = M[temp+2];   !    .
    R7 = R7+2;             !    .
    char = M[R7]<7:0>;     !    .
    R7 = R7+2;             !    .
    temp = M[R7];          !    .
    mask = M[temp]<7:0>;   !    .
    table.adr = M[temp+2];!     .
    R7 = R7+2;             !    .

    found = 0;
    while (src.len nequ 0) and (found eqlu 0) do
             if (M[table.adr+M[src.adr]] and mask) eqlu 0 then
                 begin
                 src.len = src.len-1;
                 src.adr = src.adr+1
                 end
             else found = 1;

    R0 = src.len;
    R1 = src.adr;
    R4 = 0<15:8>@mask;     ! SCANC only
    R5 = table.adr;        !    .

    N = R0<15>;
    Z = R0 eqlu 0;
    V = 0;
    C = 0;
```

Examples:

1. Find Next Digit - Register Form

```
        MOV     STR.DSCR,R0     ; string to scan
        MOV     STR.DSCR+2,R1
        MOV     #1,R4           ; mask for char set
        MOV     #TAB,R5         ; character set table
```

```
        SCANC                   ; scan string for digits
        BNE     DIGIT           ; digit found
        BEQ     NODIGIT         ; string had no digits

TAB:.BYTE   Ø                   ; ASCII ØØØ
    .BYTE   Ø                   ; ASCII ØØ1
    .BYTE   Ø                   ; ASCII ØØ2
                                    .
                                    .
                                    .
    .BYTE   1                   ; ASCII Ø6Ø = 'Ø'
    .BYTE   1                   ; ASCII Ø61 = '1'
    .BYTE   1                   ; ASCII Ø62 = '2'
    .BYTE   1                   ; ASCII Ø63 = '3'
    .BYTE   1                   ; ASCII Ø64 = '4'
    .BYTE   1                   ; ASCII Ø65 = '5'
    .BYTE   1                   ; ASCII Ø66 = '6'
    .BYTE   1                   ; ASCII Ø67 = '7'
    .BYTE   1                   ; ASCII Ø7Ø = '8'
    .BYTE   1                   ; ASCII Ø71 = '9'
    .BYTE   Ø                   ; ASCII Ø72
    .BYTE   Ø                   ; ASCII Ø73
                                    .
                                    .
                                    .
    .BYTE   Ø                   ; ASCII 377
```

2. Find Next Digit - In-line Form

```
        SCANCI                  ; scan
        .WORD   SRC.DSCR.PTR    ; ptr to src descriptor
        .WORD   SET.DSCR.PTR    ; ptr to char set dscr
        BNE     DIGIT           ; digit found
        BEQ     NODIGIT         ; string had no digits
```

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that no characters in the set were found. The original source character string descriptor is returned in RØ-Rl.

2. The source character string and character set table may overlap in any way.

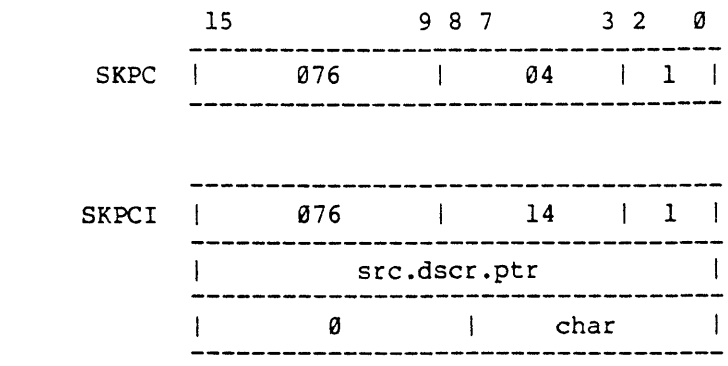
3. A test for success is BNE; a test for failure is BEQ.

4. The condition codes will be set as if this instruction were followed by TST RØ.

5. The effect of the instruction is unpredictable if the entire 256 byte character set table is not in readable memory.

5.21 SKPC / SKPCI - Skip Character

Format:



Operation:

Search source character string until a character other than the search character is found.

Condition Codes:

The condition codes are based on the final contents of RØ.

N: set if RØ<15> set; cleared otherwise
Z: set if RØ=Ø; cleared otherwise
V: cleared
C: cleared

Suspendability:

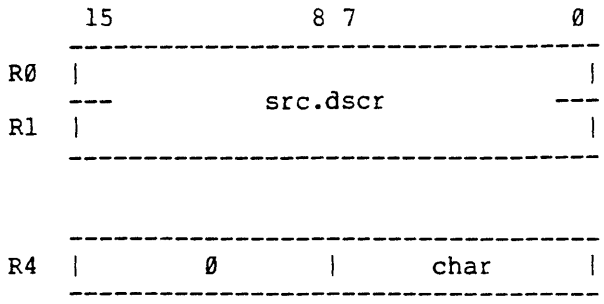
This instruction is potentially suspendable.

Description:

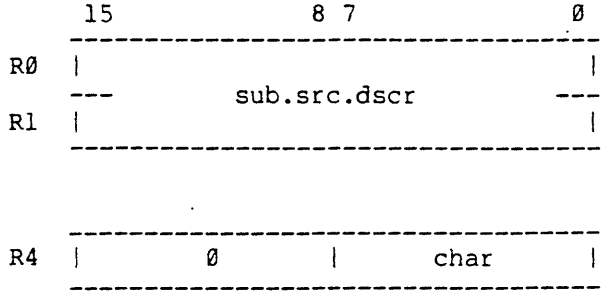
The source character string is searched from most significant to least significant character until the first occurrence of a character which is not the search character. A character string descriptor is returned in RØ-R1 which represents the portion of the source character string beginning which the most significant character which was not equal to the search character. If the source character string contains only characters equal to the search character, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in RØ.

Register Form - SKPC
-------- ---- - ----

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in RØ-Rl, the search character is placed in R4<7:Ø>, and R4<15:8> must be zero:

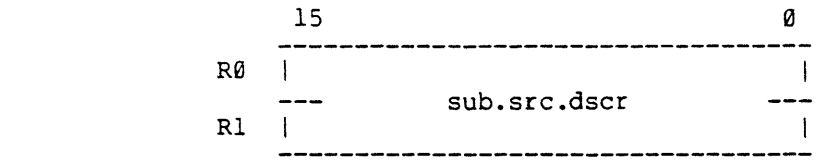


When the instruction is completed, RØ-Rl contain a character string descriptor which represents the sub-string of the source character string beginning with the most significant character which was not equal to the search character:



In-line Form - SKPCI
------- ---- - -----

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, and a word whose low order half contains the search character and whose high order half must be zero. When the instruction is completed, RØ-Rl contain a character string descriptor which represents the sub-string of the source character string beginning with the most significant character which was not equal to the search character. R2-R6 are unchanged:



Formal Description:

```
src.len = R0;          ! SKPC only
src.adr = R1;          !    .
char = R4<7:0>;        !    .

temp = M[R7];          ! SKPCI only
src.len = M[temp];     !    .
src.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
char = M[R7]<7:0>;     !    .
R7 = R7+2;             !    .

found = 1;
while (src.len nequ 0) and (found eqlu 1) do
         if M[src.adr] eqlu char then
             begin
             src.len = src.len-1;
             src.adr = src.adr+1
             end
         else found = 0;

R0 = src.len;
R1 = src.adr;
R4 = 0<15:8>@char;     ! SKPC only

N = R0<15>;
Z = R0 eqlu 0;
V = 0;
C = 0;
```

Examples:

1. Skip Leading Spaces - Register Form

```
        MOV     STR.DSCR,R0     ; string to search
        MOV     STR.DSCR+2,R1
        MOV     #' ,R4          ; space character
        SKPC                    ; skip
        BEQ     BLANK           ; line was blank
```

2. Skip Leading Spaces - In-line Form

```
SKPCI                       ; skip
.WORD   SRC.DSCR.PTR        ; ptr to src descriptor
.WORD   '                   ; space character
BEQ     BLANK               ; line was blank
```

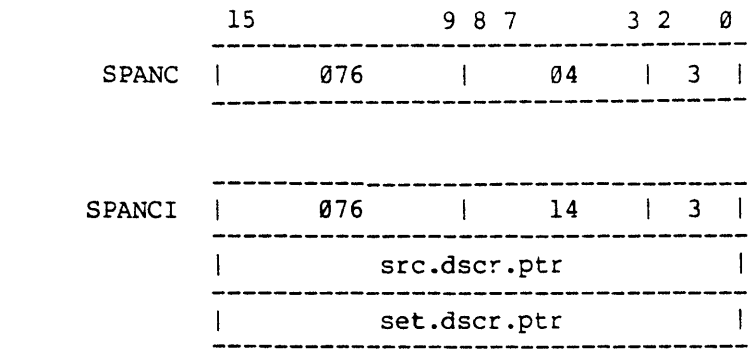
Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating the character string only contained search characters. The original source character string descriptor is returned in R0-R1.

2. The condition codes will be set as if this instruction were followed by TST R0.

5.22 SPANC / SPANCI - Span Character

Format:



Operation:

Search source character string for a character which is not a a member of the character set.

Condition Codes:

The condition codes are based on the final contents of R0.

N: set if R0<15> set; cleared otherwise
Z: set if R0=0; cleared otherwise
V: cleared
C: cleared

Suspendability:

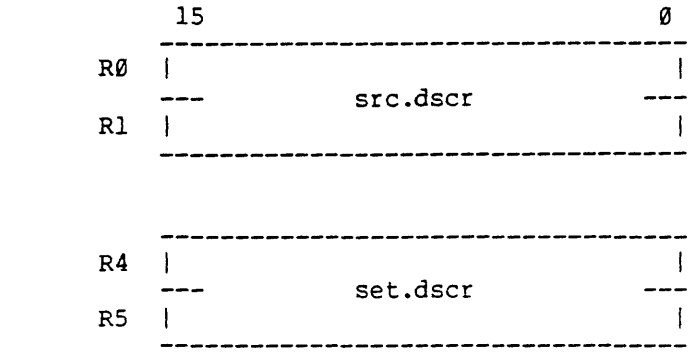
This instruction is potentially suspendable.

Description:

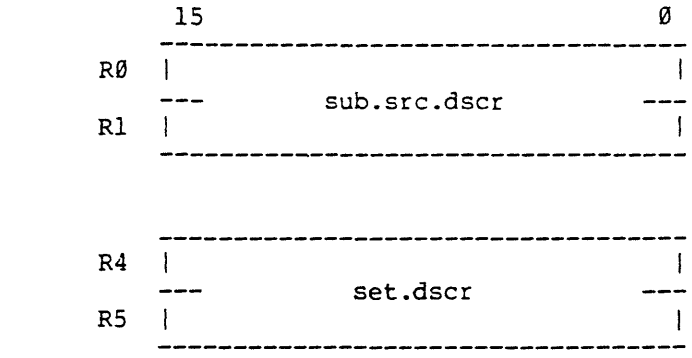
The source character string is searched from most significant to least significant character until the first occurrence of character which is not a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the character which is not a member of the character set. If the source character string contains only characters which are in the character set, the instructions return a vacant character string descriptor with an address one greater than that of the least significant character of the source character string. The condition codes reflect the resulting value in R0.

Register Form - SPANC
-------- ---- - -----

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in RØ-Rl, and the character set descriptor is placed in R4-R5:

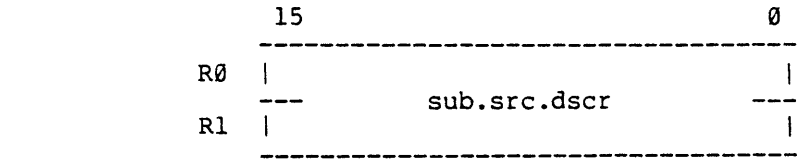


When the instruction is completed, RØ-Rl contain a character string descriptor which represents the sub-string of the source character string beginning with the character which is not a member of the character set:



In-line Form - SPANCI
------- ---- - ------

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, and a word address pointer to a two word character set descriptor. When the instruction is completed, RØ-Rl contain a character string descriptor which represents the sub-string of the source character string beginning with the character which is a member of the character set. R2-R6 are unchanged:



Formal Description:

```
src.len = R0;          ! SPANC only
src.adr = R1;          !    .
mask = R4<7:0>;        !    .
table.adr = R5;        !    .

temp = M[R7];          ! SPANCI only
src.len = M[temp];     !    .
src.adr = M[temp+2];   !    .
R7 = R7+2;             !    .
char = M[R7]<7:0>;     !    .
R7 = R7+2;             !    .
temp = M[R7];          !    .
mask = M[temp]<7:0>;   !    .
table.adr = M[temp+2];!     .
R7 = R7+2;             !    .

found = 1;
while (src.len nequ 0) and (found eqlu 1) do
         if (M[table+M[src.adr]] and mask) nequ 0 then
             begin
             src.len = src.len-1;
             src.adr = src.adr+1
             end
         else found = 0;

R0 = src.len;
R1 = src.adr;
R4 = 0<15:8>@mask;     ! SPANC only
R5 = table.adr;        !    .

N = R0<15>;
Z = R0 eqlu 0;
V = 0;
C = 0;
```

Examples:

1. Pass Tabs and Blanks - Register Form

```
        MOV     STR.DSCR,R0     ; string to scan
        MOV     STR.DSCR+2,R1
        MOV     #2,R4           ; character set mask
        MOV     #TAB,R5         ; character set table
        SPANC                   ; span
        BNE     FOUND           ; printing char found
        BEQ     EMPTY           ; string contained only
                                ; tabs and spaces

    ;
    ; The following table can be combined with the one
    ; in the SCANC example.
    ;

    TAB:.BYTE   0               ; ASCII 000
        .BYTE   0               ; ASCII 001
        .BYTE   0               ; ASCII 002
                                    .
                                    .
                                    .
        .BYTE   2               ; ASCII 011 = TAB
        .BYTE   0               ; ASCII 012
        .BYTE   0               ; ASCII 013
                                    .
                                    .
                                    .
        .BYTE   2               ; ASCII 040 = SPACE
        .BYTE   0               ; ASCII 041
        .BYTE   0               ; ASCII 042
                                    .
                                    .
                                    .
        .BYTE   0               ; ASCII 377
```

2. Pass Tabs and Blanks - In-line Form

```
        SPANCI                  ; scan
        .WORD   SRC.DSCR.PTR    ; ptr to src descriptor
        .WORD   SET.DSCR.PTR    ; ptr to char set dscr
        BNE     FOUND           ; printing char found
        BEQ     EMPTY           ; string contained only
                                ; tabs and spaces
```

Notes:

1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating that only characters in the set were found. The original source character string descriptor is returned in RØ-Rl.

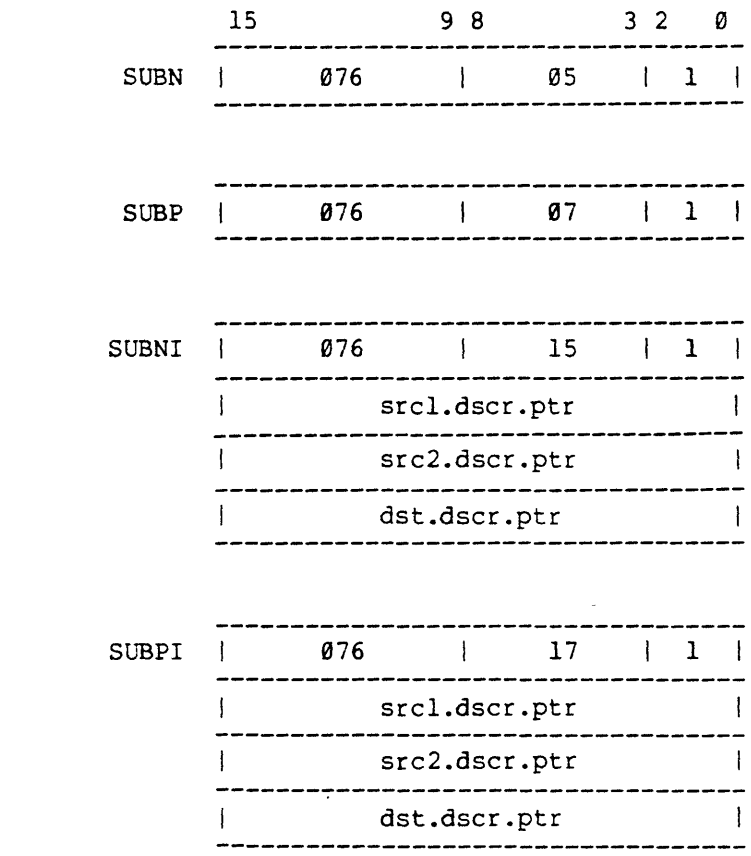
2. The source character string and character set table may overlap in any way.

3. The condition codes will be set as if this instruction were followed by TST RØ.

4. The effect of the instruction is unpredictable if the entire 256 byte character set table is not in readable memory.

5.23 SUBN / SUBP / SUBNI / SUBPI - Subtract Decimal

Format:



Operation:

dst <- src2 - src1

Condition Codes:

N: set if dst<0; cleared otherwise
Z: set if dst=0; cleared otherwise
V: set if dst can not contain all significant digits of the result; cleared otherwise
C: cleared

Suspendability:

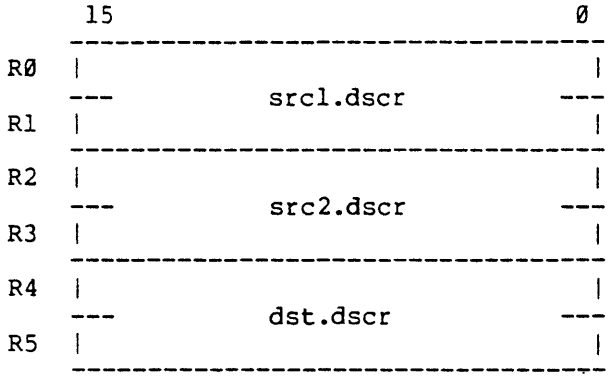
This instruction is potentially suspendable.

Description:

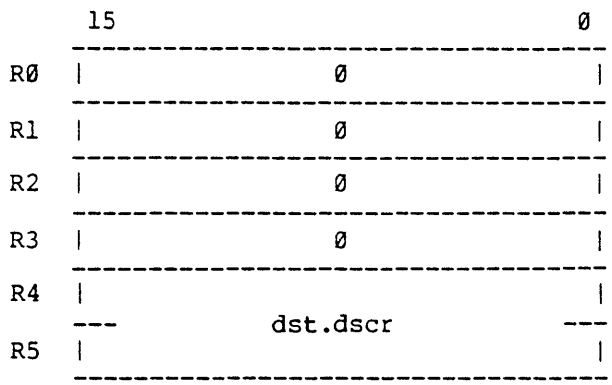
Srcl is subtracted from src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - SUBN and SUBP

When the instruction starts, the operands must have been placed in the general registers. The first source descriptor is placed in R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:



When the instruction is completed, the source descriptor registers are cleared:



In-line Form - SUBNI and SUBPI

Each word address pointer which follows the opcode word in the instruction stream refers to a two word decimal string descriptor. R0-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Three address subtract - Register Form

```
        MOV     SRC1.DSCR,R0    ; subtrahend descriptor
        MOV     SRC1.DSCR+2,R1
        MOV     SRC2.DSCR,R2    ; minuend descriptor
        MOV     SRC2.DSCR+2,R3
        MOV     DST.DSCR,R4     ; difference descriptor
        MOV     DST.DSCR+2,R5
        SUBN / SUBP             ; subtract
        BVS     OVERFLOW        ; check for error
        BLT     NEGATIVE        ; negative destination
        BEQ     EQUAL           ; zero destination
        BGT     GREATER         ; positive destination
```

2. Three address subtract - In-line Form

```
        SUBNI / SUBPI           ; subtract
        .WORD   SRC1.DSCR.PTR   ; ptr to sub descriptor
        .WORD   SRC2.DSCR.PTR   ; ptr to min descriptor
        .WORD   DST.DSCR.PTR    ; ptr to dif descriptor
        BVS     OVERFLOW        ; check for error
        BLT     NEGATIVE        ; negative destination
        BEQ     EQUAL           ; zero destination
        BGT     GREATER         ; positive destination
```

3. Two address subtract - Register Form

```
        MOV     SRC.DSCR,R0     ; subtrahend descriptor
        MOV     SRC.DSCR+2,R1
        MOV     DST.DSCR,R2     ; minuend descriptor
        MOV     DST.DSCR+2,R3
        MOV     R2,R4           ; difference descriptor
        MOV     R3,R5
        SUBN / SUBP             ; subtract
        BVS     OVERFLOW        ; check for error
        BLT     NEGATIVE        ; negative destination
        BEQ     EQUAL           ; zero destination
        BGT     GREATER         ; positive destination
```

4. Two address subtract - In-Line Form

```
SUBNI / SUBPI           ; subtract
.WORD   SRC.DSCR.PTR    ; ptr to sub descriptor
.WORD   DST.DSCR.PTR    ; ptr to min descriptor
.WORD   DST.DSCR.PTR    ; ptr to dif descriptor
BVS     OVERFLOW        ; check for error
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

Notes:

1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified dat type.

2. Source strings may overlap the destination string only if all corresponding digits of the strings are in coincident bytes in memory.

APPENDIX C
CIS ABBREVIATIONS

Abbreviation		Definition
ADR	–	Address
ALU	–	Arithmetic logic unit
AREG	–	"A" register (of BCD path)
B	–	Borrow
BR	–	Bus request
BREG	–	"B" register (of BCD path)
C	–	Carry (condition code)
C/B	–	Carry/borrow bit
CC	–	Condition code
CIS	–	Commercial instruction set
CISP	–	CIS processor
CISPW	–	CIS scratch pad write
CISS	–	CIS status
CNTL	–	Control
CPC	–	CIS program counter
DESCR	–	Descriptor
DST	–	Destination
DT	–	Data type
FNCT	–	Function
FPLA	–	Field programmable logic array
G	–	Carry generate
GPR	–	General purpose register
IBUF	–	Input buffer
INST	–	Instruction
IR	–	Instruction register
L2dr	–	Load 2 descriptor
L3dr	–	Load 3 descriptor
LS	–	Local store
m	–	Default value
MPC	–	Microprogram counter
N	–	Negative (condition code)
OVR	–	Overflow
P	–	Carry propagate
PSW	–	Processor status word
SRC	–	Source
V	–	Overflow (condition code)
Z	–	Zero (condition code)

APPENDIX D
CISP MNEMONICS

Microword	Definition
ALUDST	ALU destination field ⟨61:59⟩
ALUFTN	ALU function field ⟨58:56⟩
ALUSRC	ALU source field ⟨55:53⟩
APORT	"A" address field of 2901A RAM
BCDMX1	BCD multiplexer 1 field ⟨29:28⟩
BCDMX3	BCD multiplexer 3 field ⟨31:30⟩
BCDOP	BCD operation field ⟨33:32⟩
BMUX	B multiplexer field ⟨35:34⟩
BPORT	"B" address field of 2901A RAM
CISSPW	CIS scratch pad write field ⟨71:70⟩
CON2	Control 2 field ⟨27:25⟩
CON3	Control 3 field ⟨24:21⟩
CON4	Control 4 field ⟨20:16⟩
CONBR1	Conditional branch 1 field ⟨5:2⟩
CONBR2	Conditional branch 2 field ⟨9:6⟩
CONST	Constant field ⟨40:38⟩
ENCB	Enable carry/borrow bit ⟨0⟩
ENCIS	Enable CIS bit ⟨1⟩
ENIB	Disable input buffer bit ⟨48⟩
ENOB	Enable output buffer bit ⟨47⟩
ENSNIN	Enable sign input bit ⟨37⟩
ENSNOU	Enable sign output bit ⟨36⟩
INEN	Input enable bit ⟨51⟩
LBYTE	Low byte enable bit ⟨46⟩
MPC	Microprogram counter field ⟨15:10⟩
SALUI	Select ALU input bit ⟨52⟩
SHFTC	Shift control field ⟨63:62⟩
SHFTIN	Shifted in bit ⟨64⟩
SWAP	Swap bytes in a word or in a data string ⟨50:49⟩

Reader's Comments

KE44-A CISP User's Guide
EK-KE44A-UG-002

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our publications.

What is your general reaction to this manual? In your judgement is it complete, accurate, well organized, well written, etc? Is it easy to use? _____

_____

_____

_____

What features are most useful? _____

_____

_____

_____

What faults or errors have you found in the manual? _____

_____

_____

_____

Does this manual satisfy the need you think it was intended to satisfy? _____

Does it satisfy *your* needs? _____ Why? _____

_____

_____

_____

☐ Please send me the current copy of the *Technical Documentation Catalog*, which contains information on the remainder of DIGITAL's technical documentation.

Name _____ Street _____
Title _____ City _____
Company _____ State/Country _____
Department _____ Zip _____

Additional copies of this document are available from:

Digital Equipment Corporation
Accessories and Supplies Group
Cotton Road
Nashua, NH 03060

Attention *Documentation Products*
Telephone 1-800-258-1710

Order No. EK-KE44A-UG-002

Fold Here

Do Not Tear – Fold Here and Staple

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 33 MAYNARD, MA

POSTAGE WILL BE PAID BY ADDRESSEE

Digital Equipment Corporation
Educational Services Development and Publishing
1925 Andover Street
Tewksbury, Massachusetts 01876