EK-KE44A-TM-001

KE44-A CISP Technical Manual

Prepared by Educational Services
of
Digital Equipment Corporation

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

CONTENTS

FIGURES

TABLES

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 significantly extends the capability of the PDP-11/44 computer in the area of commercial data processing. The KE44-A option is installed in the PDP-11/44 cabinet.

CIS-specific abbreviations used in this manual are listed in Appendix C. Appendix D lists the CIS microword mnemonics.

1.2 GENERAL DESCRIPTION

1.2.1 Commercial Instruction Set

The CIS is a series of instructions for manipulating byte strings in order 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 076 nnn.

1.2.2 Suspension (Interrupt)

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, "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, executing one or more new CIS instructions during the period of suspension. At the end of this interrupt routine, control is returned to the KE44-A for completion of the suspended instructions. 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.

1.2.3 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 CIS program counter (CPC) field ⟨87:76⟩ of each microword is coded with the address of the next microword.

1.2.4 Hardware Description
The main hardware elements of the KE44-A are a control store module and a data path module. 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 (PC) and Microprocessor Code (MPC) Addressing logic
2. Binary data path logic
3. Decimal data path logic
4. Status Information and Condition Code Generation logic

These sections are described in detail in Chapter 4.

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
INTERFACING

2.1 GENERAL

The KD11-Z CPU loads commercial instructions and operands into the CISP KE44-A. After the CISP executes the requested operation, the CPU reads the results and stores them in memory. Figure 2-1 shows the KE44-A/CPU interface lines; Table 2-1 describes the interface signals.

NOTE

The KE44-A does not directly interface with the UNIBUS, but is connected to the KD11-Z via a bus that is separate from the UNIBUS and uses the KD11-Z microcode for data transfers to and from memory.

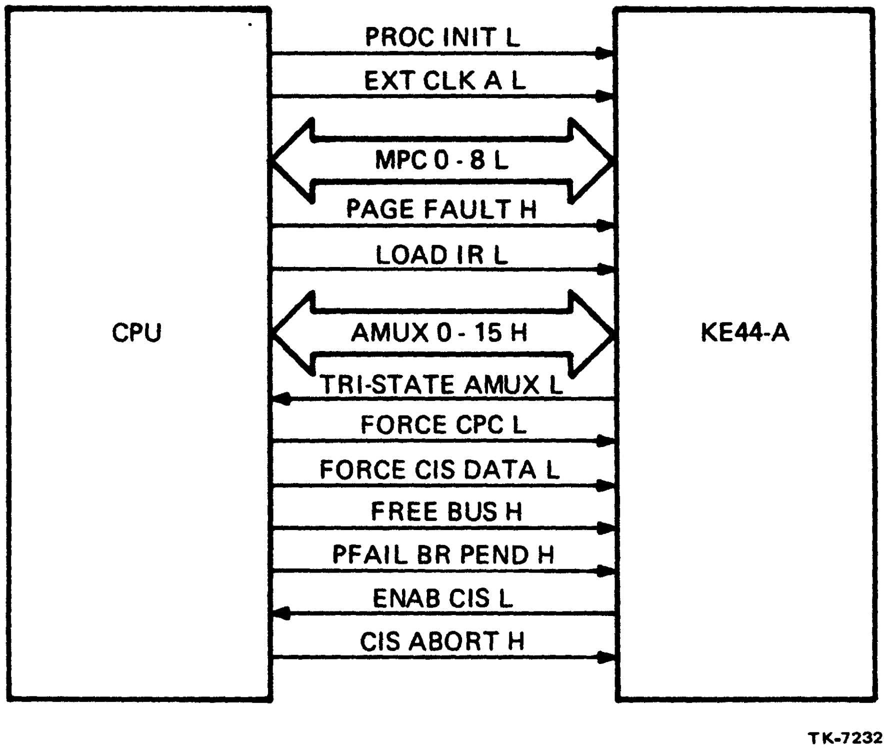


Figure 2-1 KE44-A/CPU Interface Lines

Table 2-1 KE44-A/CPU Interface Line Definitions

Mnemonic	Signal Flow	Function
MPC ⟨00:08⟩ L	Bidirectional	Microprogram address lines. Used to sequence the CPU through the microprogram. Derived from KE44-A microcode. Cannot be altered by CPU.
AMUX ⟨00:15⟩ L	Bidirectional	Data lines used to transfer instructions and operands between CPU and KE44-A.
ENAB CIS L	KE44-A to CPU	Forces CPU to a service state after the completion of a CIS instruction; i.e., when a low-to-high signal transition occurs.
CIS ABORT H	CPU to KE44-A	Clears CIS CPC line when an abort condition exists in CPU.
PROC INIT L	CPU to KE44-A	CPU initialize. Used to initialize status registers in KE44-A.
LOAD IR L	CPU TO KE44-A	Cause KE44-A to load its instruction register (IR) from AMUX lines.
TRI-STATE AMUX L	KE44-A to CPU	Causes CPU to remove data from AMUX lines. Turns off the KD11-Z drivers, thus enabling KE44-A access to the AMUX lines.
PFAIL BR PEND H	CPU to KE44-A	When high, indicates that an interrupt needs servicing. Used by the KE44-A to suspend instructions in the middle of execution.
PAGE FAULT H	CPU TO KE44-A	If high, indicates that a page of memory cannot be written into. This signal is generated by probing, rather than writing to the page.
FORCE CIS DATA L	CPU to KE44-A	Console-generated signal for monitoring MBUS data via the AMUX lines.
FREE BUS H	CPU to KE44-A	Console-generated signal that tri-states all maintenance that drive the AMUX lines.
EXT CLK A L	CPU to KE44-A	CPU signal that clocks the control word through the control logic.
FORCE CPC L	CPU to KE44-A	Console-generated signal for monitoring the CPC lines via the AMUX lines.

2.2 INITIAL OPERATION

Initially, the CPU fetches an instruction from memory and decodes it in the CPU and CIS. During this fetch, LOAD IR L is asserted to load the CIS IR. Any instruction with an op code of 0760xx or 0761xx is a commercial instruction and requires the use of the KE44-A to process. The CIS next asserts 740 on the MPC 0–8 microprocessor code bus (MPC bus). Since CIS instructions are only recognized by the KE44-A option, the assertion of MPC 740 is required to prevent the CPU from trapping on an illegal instruction. MPC 740 is decoded by the KD11-Z to set up the CIS processor for an operation in the next CPU cycle. Concurrently with this decoding, a CPC (CIS program counter) address is asserted to the 88-bit control store of the KE44-A. This control word is clocked by EXT CLK A L from the CPU.

2.3 MICROCODE GENERATION

A series of microcodes is generated in the KE44-A to control microprocessor operation during each instruction. During CIS operation, the KE44-A informs the CPU (via a microcode asserted on the MPC 0–8 lines) whenever data can be read from the AMUX 0–15 lines. The KE44-A also sends the CPU a TRI STATE AMUX L signal that enables it to read data from the AMUX lines. The CPU then stores this data and continues operation.

NOTE

Chapter 3 has been duplicated directly from DECSTD168-PDP-11 Extended Instructions.

CHAPTER 3
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:

```
     7          0
    --------------
  A |   char     |
    --------------
```

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 0 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:

```
                 15                            0
                 ---------------------------------
  Rx      ptr    |               n               |
      or         ---------------------------------
  Rx+1    ptr+2  |               A               |
                 ---------------------------------
```

The following figure shows the character string in memory:

```
                 7           0
                ---------------
             A  |most sig char|
                ---------------

                ---------------
           A+1  |             |
                ---------------
                       .
                       .
                       .
                ---------------
         A+n-1  |least sig chr|
                ---------------
```

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 00000001(2), 00000010(2), 00000100(2), 00001000(2), 00010000(2), 00100000(2), 01000000(2) and 10000000(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 00000001(2) appearing in the table corresponds to the character subset of all upper case alphabetic characters, 00000010(2) appearing in the table corresponds to the character subset of all lower case alphabetic characters, and 00000100(2) appearing in the table corresponds to the decimal digits, then using the mask 00000011(2) with this table specifies the character set of all alphabetic characters, and using the mask 00000111(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.

```
                     15             8 7              Ø
                    -----------------------------------
        Rx     ptr  |       Ø        |      mask      |
           or       -----------------------------------
        Rx+1   ptr+2 |           table address         |
                    -----------------------------------
```

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 0. 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 <4:0>	- Number of digits specified as an unsigned binary integer.
data type <14:12>	- Specifies which decimal data type representation is used.

Second Word:

address <15:0>	- 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':

```
                      15 14  12 11        5 4       0
                      ------------------------------------
Rx        ptr         | 0|   T  |       0    |    L     |
     or               ------------------------------------
Rx+1      ptr+2       |                 A                |
                      ------------------------------------
```

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

000	signed zoned
001	unsigned zoned
010	trailing overpunch
011	leading overpunch
100	trailing separate
101	leading separate
110	-- reserved by the architecture
111	-- reserved by the architecture

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

ØØØ	-- reserved by the architecture
ØØ1	-- reserved by the architecture
Ø1Ø	-- reserved by the architecture
Ø11	-- reserved by the architecture
1ØØ	-- reserved by the architecture
1Ø1	-- reserved by the architecture
11Ø	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:Ø> and the least significant digit in bits <7:4>.

Signed Packed Strings -

The preferred positive sign designator is 11ØØ(2); alternate positive sign designators are 1Ø1Ø(2), 111Ø(2) and 1111(2). The preferred negative sign designator is 11Ø1(2); the alternate negative sign designator is 1Ø11(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	11ØØ(2)	1Ø1Ø(2) 111Ø(2) 1111(2)
negative	11Ø1(2)	1Ø11(2)
unsigned	1111(2)	

For other than the least significant byte, bytes contain two consecutive digits -- the one of lower significance in bits <3:Ø> 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:Ø> of the lowest addressed byte; bits <7:4> of this byte must be zero for source strings, and are cleared to ØØØØ(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:

```
                    7    4 3    0
                    --------------
                A   |  msd |      |
                    --------------

                    --------------
            A+1     |      |      |
                    --------------
                           .
                           .
                           .
                    --------------
  A+[length/2]      | lsd  | sign |
                    --------------
```

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

```
                    7    4 3    0
                    --------------
                A   |  0   | msd  |
                    --------------

                    --------------
            A+1     |      |      |
                    --------------
                           .
                           .
                           .
                    --------------
  A+[length/2]      | lsd  | sign |
                    --------------
```

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:

```
                    7    4 3    0
                    --------------
                A   |  0   | sign |
                    --------------
```

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.

```
          7     4 3    0
          --------------
   A      |      | msd  |
          --------------

          --------------
   A+1    |      |      |
          --------------
                 .
                 .
                 .
          --------------
   A+n-1  | sign | lsd  |     'sign' is present only
          --------------      signed zoned strings
```

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 0 to 31(10) digits.
2. The low order nibble of each byte is in the range 0000(2) to 1001(2).
3. Signed Zoned Strings - The high order nibble of the least significant byte is either 0011(2) or 0111(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 <3:0>). 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 0011(2) in the high order nibble of all bytes which do not contain the sign. 0011(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 '0' to '9', '[', '?', ']', '!' and ':'.

OVERPUNCH SIGN/DIGIT BYTE:

Overpunch Sign/Digit	Preferred Designator	Alternate Designators
+0	01111011(2)	00110000(2), 01011011(2), 00111111(2)
+1	01000001(2)	00110001(2)
+2	01000010(2)	00110010(2)
+3	01000011(2)	00110011(2)
+4	01000100(2)	00110100(2)
+5	01000101(2)	00110101(2)
+6	01000110(2)	00110110(2)
+7	01000111(2)	00110111(2)
+8	01001000(2)	00111000(2)
+9	01001001(2)	00111001(2)
-0	01111101(2)	01011101(2), 00100001(2), 00111010(2)
-1	01001010(2)	
-2	01001011(2)	
-3	01001100(2)	
-4	01001101(2)	
-5	01001110(2)	
-6	01001111(2)	
-7	01010000(2)	
-8	01010001(2)	
-9	01010010(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:

```
          7    4 3    0
          ---------------
    A     |       | msd  |
          ---------------

          ---------------
    A+1   |       |      |
          ---------------
                 .
                 .
                 .
          ---------------
  A+n-1   | sign and lsd|
          ---------------
```

The following is a leading overpunch string:

```
           7    4 3     0
           --------------
A          | sign and msd|
           --------------

           --------------
A+1        |      |      |
           --------------
                  .
                  .
                  .
           --------------
A+n-1      |      | lsd  |
           --------------
```

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 0 to 31(10) digits.

2. The low order nibble of each digit byte is in the range 0000(2) to 1001(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 <3:0>). 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:

```
          7    4 3    0
          ---------------
    A     |      | msd  |
          ---------------

          ---------------
    A+1   |      |      |
          ---------------
                 .
                 .
                 .
          ---------------
    A+n-1 |      | lsd  |
          ---------------

          ---------------
    A+n   |    sign     |
          ---------------
```

The following is a leading separate string:

```
                    7   4 3    0
                    --------------
          A-1       |    sign    |
                    --------------

                    --------------
            A       |      | msd |
                    --------------

                    --------------
          A+1       |      |     |
                    --------------
                           .
                           .
                           .
                    --------------
        A+n-1       |      | lsd |
                    --------------
```

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:

```
                    7          0
                    --------------
            A       |    sign    |
                    --------------
```

The following is a zero length leading separate string:

```
                    7          0
                    --------------
          A-1       |    sign    |
                    --------------

            A
```

A valid separate string is characterized by:

1. A length from 0 to 31(10) digits.
2. The low order nibble of each digit byte is in the range 0000(2) to 1001(2).
3. The sign byte is either 00100000(2), 00101011(2) or 00101101(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:

```
          15 14                            Ø
         -------------------------------------
     R2  |s |            high                 |
         -------------------------------------
     R3  |               low                  |
         -------------------------------------
```

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

```
          15 14                            Ø
         -------------------------------------
    ptr  |               low                  |
         -------------------------------------
  ptr+2  |s |            high                 |
         -------------------------------------
```

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.

CHAPTER 4
THEORY OF OPERATION

4.1 2901A MICROPROCESSOR SLICER

A functional block diagram of the KE44-A (Figure 4-1) shows the use of 2901A in the binary data path. The 2901A has four 4-bit microprocessor slices that are configured for carry lookahead and external shift control in the 16-bit data path (Figure 4-2). The principal elements in each of the identical 2901As are: 1) a 16-location RAM, 2) a high-speed ALU, and 3) a separate, shiftable holding register called the Q-register. The RAM locations are used as KE44 working registers. The ALU, in conjunction with the working registers and the Q-register, performs the arithmetic and logical functions necessary to implement the macroinstruction set. Data enters the 2901As from the 2901A D bus; 2901A output data is transmitted on the 2901A Y bus. The output data is from either the 2901A ALU or the contents of a 2901A RAM (working register) location.

4.1.1 2901A RAM

In the KE44-A, the RAM of the 2901As is the scratch pad area where the results of the arithmetic and logical operations can be temporarily stored. RAM contents are read into the ALU in response to microcode control signals received from the control store logic. Since each of the four 2901A microprocessors comprising this RAM has a 16 $\times$ 4-bit slice, the combination yields a total RAM capacity of sixteen 16-bit words.

The data in any of these words can be read from the A port via the 4-bit RAM APORT 0–3 H address line inputs. If the same address is applied to both A and B address lines, identical data appears at both RAM output ports. The RAM A and B outputs are applied to latches. When the RAM is write-enabled, new data is written into the RAM word selected by the RAM B input PORT 0–3 H. The RAM input data is received from a 3-input multiplexer (Figure 4-2). Multiplexer D0–D3 inputs from the ALU output permit the ALU result to be loaded into the RAM directly, or to be left- or right-shifted by one place.

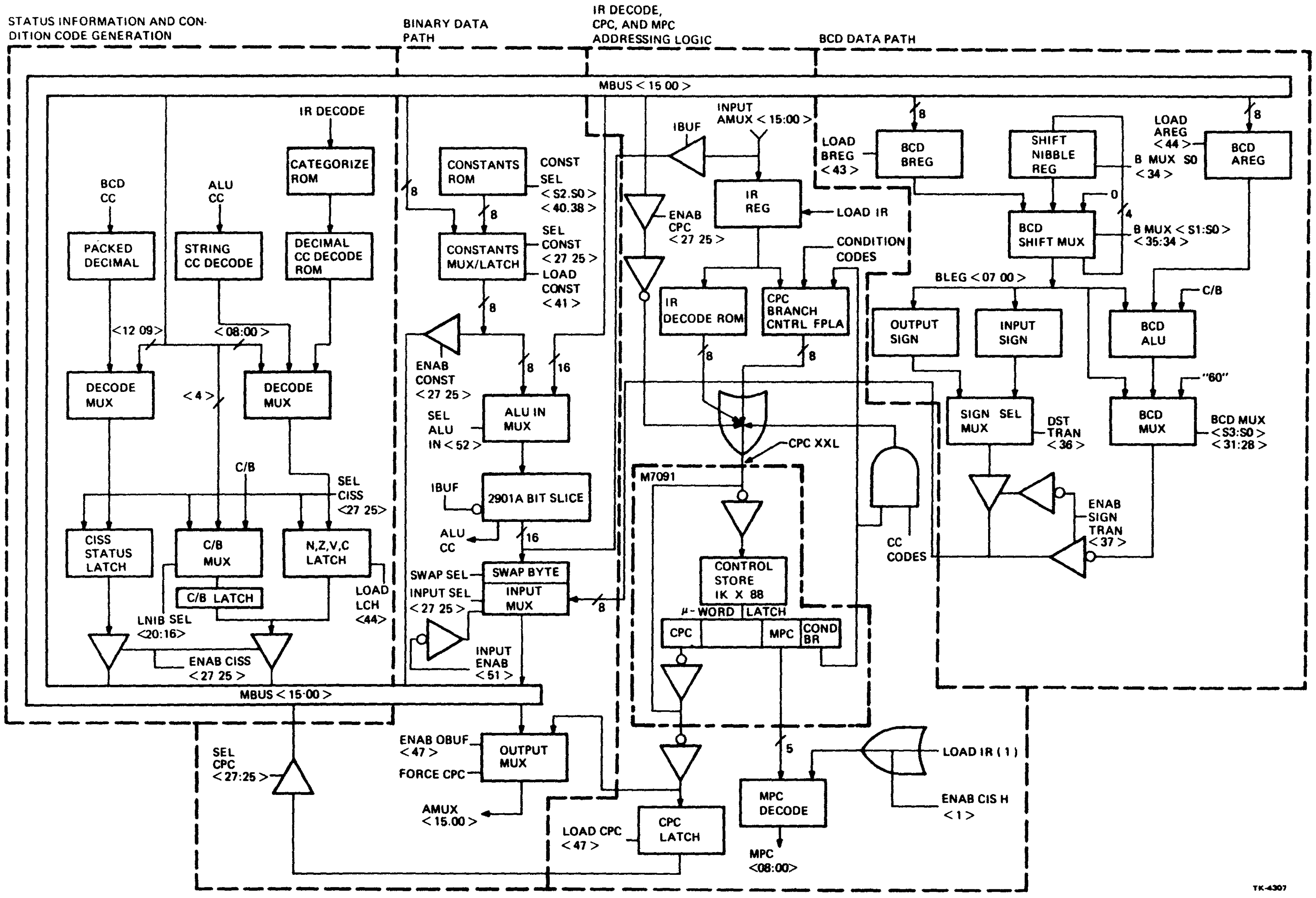


Figure 4-1 KE44-A Block Diagram

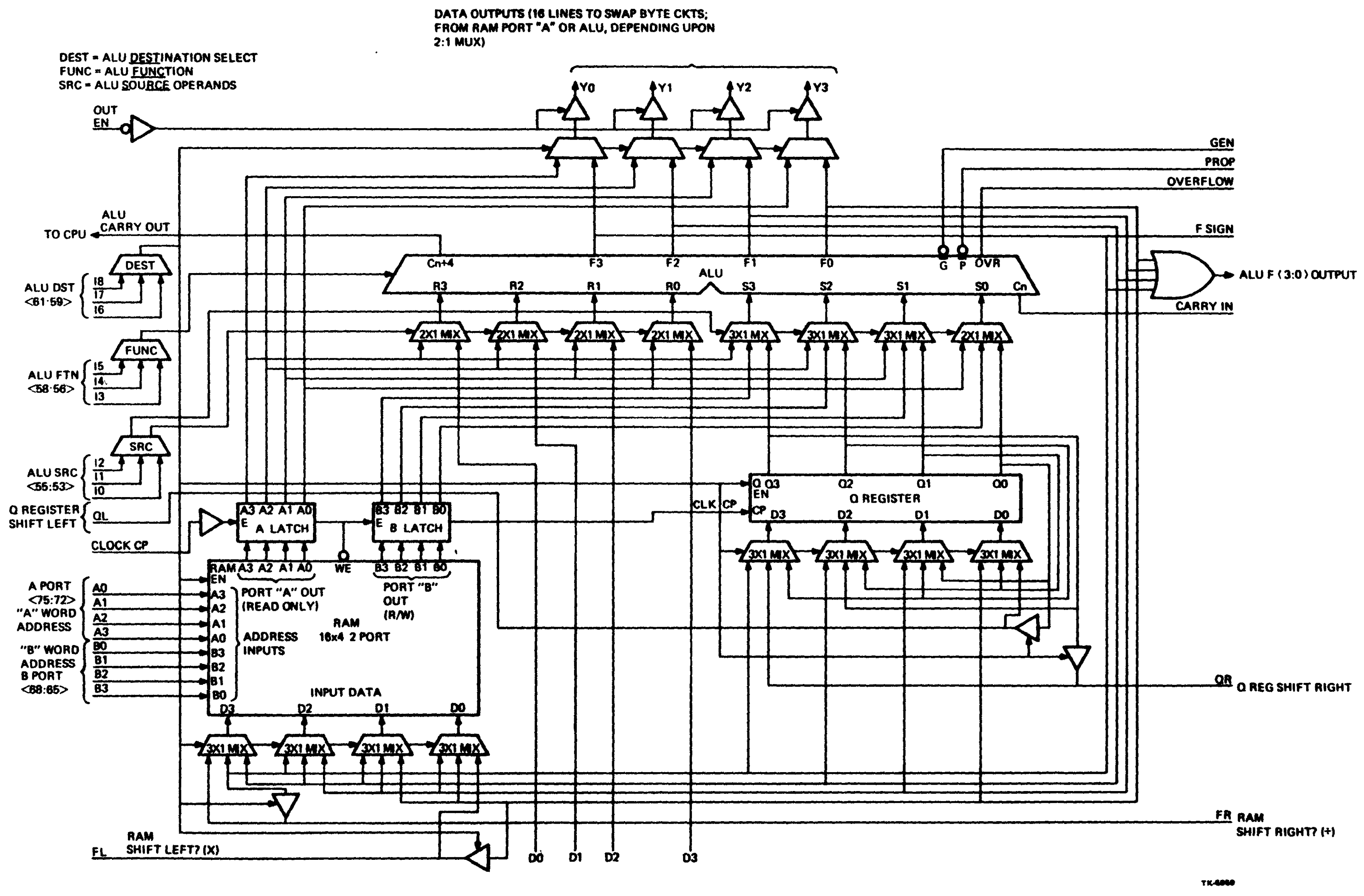


Figure 4-2 2901A Detailed Block Diagram

4.1.2 Arithmetic Logic Unit (ALU)

The ALU is the data path component that performs the arithmetic/logical operation under command of the microcode control word (Table 4-1) contained in the control store logic PROMs. ALU R inputs are from four 2-input multiplexers whose inputs are the direct data inputs D3–D0 and the A port outputs A3-A0 of the RAM. The S inputs, received via four 3-input multiplexers, include the A and B ports of the RAM and the Q-register outputs.

Decode of the ALU function (FUNC) lines I3–I5 determines the arithmetic or logical function to be performed. Decode of the ALU destination (DEST) lines I8–I6 determines which of the indicated registers the data is routed to or whether it will be a data output of the device itself. ALU output data F3–F0 can be routed to the Q-register or RAM, or placed on lines as Y3–Y0.

Table 4-1 Microcode Matrix for Source Operands and ALU Functions

	I_{210} Octal	0	1	2	3	4	5	6	7
I_{543}Octal	ALU Source / ALU Function	A,Q	A,B	O,Q	O,B	O,A	D,A	D,Q	D,O
0	C_n=L R Plus S	A+Q	A+B	Q	B	A	D+A	D+Q	D
	C_n=H	A+Q+1	A+B+1	Q+1	B+1	A+1	D+A+1	D+Q+1	D+1
1	C_n=L S Minus R	Q-A-1	B-A-1	Q-1	B 1	A 1	A-D-1	Q-D-1	-D-1
	C_n=H	Q-A	B-A	Q	B	A	A-D	Q-D	-D
2	C_n=L R Minus S	A-Q-1	A-B-1	-Q-1	-B-1	-A-1	D-A-1	D-Q-1	D-1
	C_n=H	A-Q	A-B	-Q	B	-A	D-A	D-Q	D
3	R OR S	$A \vee Q$	$A \vee B$	Q	B	A	D-A	$D \vee Q$	D
4	R AND S	$A \wedge Q$	$A \wedge B$	0	0	0	$D \wedge A$	$D \wedge Q$	0
5	R AND S	$\overline{A} \wedge Q$	$\overline{A} \wedge B$	Q	B	A	$\overline{D} \forall A$	$\overline{D} \forall Q$	0
6	R EX-OR S	$A \forall Q$	$A \forall B$	Q	B	A	$D \forall A$	$D \forall Q$	D
7	R EX-NOR S	$\overline{A \forall Q}$	$\overline{A \forall B}$	Q	B	A	$\overline{D \forall A}$	$\overline{D \forall Q}$	D

+ = Plus; - = Minus; $\vee$ = OR, $\wedge$ = AND; $\forall$ = EX OR

The ALU source operand decode performs the actual register selection. All three of these functions are controlled by ALU instructions I8–I0 from the control store logic.

The ALU can perform three binary arithmetic and five logical operations on the two input words received via the R and S inputs. Each R input is driven by a separate 2 input multiplexer and each S input from a separate 3 input multiplexer. In the KE44-A CIS, the R input multiplexer can be used to select either the RAM A port data or a direct data input consisting of constants or MBUS data. The S input multiplexer selects the Q-register output or the RAM output of port A or B. Both multiplexers have an inhibit output capability that produces a source operand of zero.

4.1.2.1 Logical and Arithmetic Functions – The ALU performs five logical and three arithmetic functions on eight source operand pairs. ALU logic functions and appropriate control bit values (source select I2–I0, and function select I5–I3) are described in Table 4-2. The carry input (C_n) has no effect on operations in logic mode but does affect operations in arithmetic mode (Table 4-3), which defines carry-in high ($C_n = 1$) and carry-in low ($C_n = 1$) for this mode.

Table 4-2 ALU Logic Mode Functions

Octal I_{543}, I_{210}	Group	Function	Octal I_{543}, I_{210}	Group	Function
40		$A \wedge Q$	74	Invert	$\overline{A}$
41		$A \wedge B$	77		$\overline{D}$
45	AND	$D \wedge A$			
46		$D \wedge Q$	62		Q
			63		B
30		$A \vee Q$	64	Pass	A
31		$A \vee B$	67		D
35	OR	$D \vee A$			
36		$D \vee Q$	32		Q
			33		B
60		$A \forall Q$	34	Pass	A
61		$A \forall B$	37		D
65	EX OR	$D \forall A$			
66		$D \forall Q$	42		0
			43		0
70		$\overline{A \vee Q}$	44	"Zero"	0
71		$\overline{A \vee B}$	47		0
75	EX NOR	$\overline{D \vee A}$			
76		$\overline{D \vee Q}$	50		$\overline{A} \wedge Q$
			51		$\overline{A} \wedge B$
72		$\overline{Q}$	55	Mask	$\overline{D} \wedge A$
73		$\overline{B}$	56		$\overline{D} \wedge Q$

Table 4-3 ALU Arithmetic Mode Functions

Octal	$C_n = 0$ (Low)		$C_n = 1$ (High)	
I_{543}, I_{210}	Group	Function	Group	Function
0 0		A+Q		A+Q+1
0 1	ADD	A+B	ADD plus	A+B+1
0 5		D+A	one	D+A+1
0 6		D+Q		D+Q+1
0 2		Q		Q+1
0 3	PASS	B	Increment	B+1
0 4		A		A+1
0 7		D		D+1
1 2		Q-1		Q
1 3	Decrement	B-1	Pass	B
1 4		A-1		A
2 7		D-1		D
2 2		-Q-1		-Q
2 3	1's Comp.	-B-1	2's Comp.	-B
2 4		-A-1	(Negate)	-A
1 7		-D-1		-D
1 0		Q-A-1		Q-A
1 1	Subtract	B-A-1	Subtract	B-A
1 5	(1's Comp.)	A-D-1	(2's Comp.)	A-D
1 6		Q-D-1		Q-D
2 0		A-Q-1		A-Q
2 1		A-B-1		A-B
2 5		D-A-1		D-A
2 6		D-Q-1		D-Q

4.1.2.2 Logical Functions for G, P, C_{N+4}, and OVR - When the microprocessor is in the add or subtract mode, signals G and P indicate carry lookahead, C_{n+4} indicates carry, and OVR indicates overflow conditions. However, OVR is not used in the CIS implementation. Table 4-4 gives the logic equations for the G, P and C_{n+4} signals for each of the eight ALU functions. The R and S inputs are the two inputs selected in accordance with Table 4-4.

4.1.3 Q-Register

The Q-register, a file loaded from the ALU, functions as a temporary storage register. Q-register output can be loaded back into itself, or shifted right or left (e.g., during convert, multiply, divide or arithmetic shift operations).

Table 4-4 P, G, C_{N+4}, and OVR Functions

I_{543}	Function	P	G	C_{n+4}	OVR
0	$R + S$	$\overline{P_3P_2P_1P_0}$	$\overline{G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0}$	C_4	$C_3 \forall C_4$
1	$S - R$	Same as $R + S$ equations, but substitute $\overline{R}_i$ for R_i in definitions			
2	$R - S$	Same as $R + S$ equations, but substitute $\overline{S}_i$ for S_i in definitions			
3	$R \vee S$	Low	$P_3P_2P_1P_0$	$\overline{P_3P_2P_1P_0} + C_n$	$\overline{P_3P_2P_1P_0} + C_n$
4	$R \wedge S$	Low	$\overline{G_3 + G_2 + G_1 + G_0}$	$G_3 + G_2 + G_1 + G_0 + C_n$	$G_3 + G_2 + G_1 + G_0 + C_n$
5	$\overline{R} \wedge S$	Low	Same as $R \wedge S$ equations, but substitute $\overline{R}_i$ for R_i in definitions		
6	$R \forall S$	Same as $\overline{R \forall S}$, but substitute $\overline{R}_i$ for R_i in definitions			
7	$\overline{R \forall S}$	$G_3 + G_2 + G_1 + G_0$	$G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1P_0$	$\overline{G_3 + P_3G_2 + P_3P_2G_1} +$ $\overline{P_3P_2P_1P_0(G_0 + \overline{C}_n)}$	$[\overline{P}_2 + \overline{G}_2\overline{P}_1 + \overline{G}_2\overline{G}_1\overline{P}_0 + \overline{G}_2\overline{G}_1\overline{G}_0C_n] \forall$ $\overline{[\overline{P}_3 + \overline{G}_3\overline{P}_2 + \overline{G}_3\overline{G}_2\overline{P}_1 + \overline{G}_3\overline{G}_2\overline{G}_1\overline{P}_0 + \overline{G}_3\overline{G}_2\overline{G}_1\overline{G}_0C_n]}$

$+$ = OR	$P_0 = R_0 + S_0$	$G_0 = R_0S_0$	$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_n$
$\vee$ = OR	$P_1 = R_1 + S_1$	$G_1 = R_1S_1$	$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_n$
$\wedge$ = AND	$P_2 = R_2 + S_2$	$G_2 = R_2S_2$	
$\forall$ = EX OR	$P_3 = R_3 + S_3$	$G_3 = R_3S_3$	

4.1.4 Bit Shifting
After data has been parallel-loaded into the microprocessor, both the Q-register and any RAM data, addressed by either A or B port, may be shifted left or right. To accomplish these shifts, the most significant bit (MSB) of each 4-bit microprocessor is connected to the least significant bit (LSB) of the adjacent, more significant, 4-bit microprocessor via a bidirectional transfer line. During a shift operation, the bit transferred out of the last 2901A (E47) is used as the final shifted-out bit.

4.1.5 Status Bits
Each 4-bit microprocessor generates two status bits, F=0 and F sign.

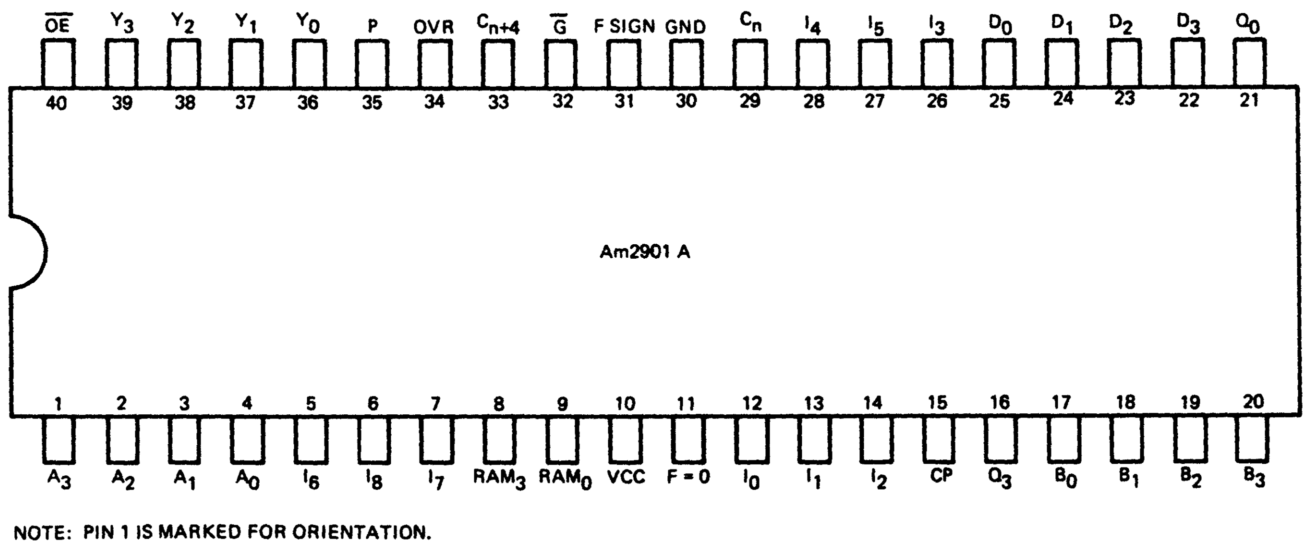
The F=0 status bit provides zero detection by indicating when the data equals zero. It is an open-collector output which "wire ORs" the two 2901As associated with each byte. Each byte, therefore, has a signal that indicates zero. These signals are CIS ALU 0–7=0 H and CIS ALU 8–15=0 H.

The F sign output is used to monitor the MSB of each 4-bit microprocessor. Only the highest nibble in each byte is monitored. The signal names are CIS ALU 07 H and CIS ALU 15 H. The F3 outputs of the low nibbles for the two remaining 4-bit microprocessors are not used.

Both status bits (CIS ALU 15 H and CIS ALU 07 H) can be monitored without enabling the output driver in the 4-bit microprocessor. Either bit can be used as a sign bit during CIS operations.

4.1.6 Carry Lookahead Logic
The 2901As use full lookahead carry logic that speeds instruction execution. Each of the four 4-bit microprocessors generates both a carry generate (G) and a carry propagate (P) output. The four pairs of G and P signals are combined in a single 745182 lookahead carry generator.

4.1.7 2901A Pin Definitions
Pin assignments for the AM2901A, 40 pin dual in-line package are shown in Figure 4-3.



NOTE: PIN 1 IS MARKED FOR ORIENTATION.

TK-7231

Figure 4-3 AM2901A Pin Connections

4.2 INSTRUCTION SUSPENSION (INTERRUPT)

The execution time for some CIS instructions will use more CPU time if longer than normal string lengths are involved. Therefore, to keep BR latencies below 35 microseconds, the CIS permits interrupt of all CIS instructions except two (L2Dr and L3Dr). The interrupt routine, called "CIS instruction suspension," allows an interrupted instruction to be restarted from the point of breakoff. This feature is important because to run an entire instruction again from its beginning would mean very long execution times. Suspension allows high priority devices to interrupt the processing sequence so that the CPU can service the interrupting device.

Any number of interrupts can be made during a CIS instruction. Also any CIS instruction can be interrupted by another CIS instruction since all the necessary information is stored on the stack and not in the CIS.

4.2.1 Steps Leading To Suspension

The CIS microcode checks for BRs at specific points in the microinstructions. Macroinstructions, a collection of microinstructions that read like English, are used to test for service. The macroinstruction used is either a "service?" or a previously defined macro which adds "service?" to it. This macro sets the CONBR2 field to a value of 07, to address a conditional branch PROM (programmable read-only memory) whose output enables E78 (74S03).

Upon receipt of a BR, the CPU asserts PFAIL BR PEND. This signal is "ANDed" with the CONBR2 field (referred to above) to assert CPC00. The CIS microcode then branches to a location corresponding to the existing CPC "ORed" with a 1. This routine is the start of a CIS "save state" operation that pushes all necessary information onto the processor stack.

In a series of instructions, the microcode also pushes the CIS status, the contents of some of the 2901A registers, and the returning CPC address onto the stack. The CIS then moves the PC address back to the beginning of the CIS instruction; PSW bit 8 is set (indicating a suspended CIS instruction), and the device interrupt service routine is entered.

4.2.2 Returning from Suspension

After the interrupt is serviced, the stack is popped, thereby returning the processor to the previous PC and processor status word (PSW). The PC used is the backed-up PC. The PSW has bit 8 set, indicating a suspended instruction.

The CIS instruction begins execution as if a suspension had not occurred. The CIS microcode tests for PSW bit 8 which, if set, causes branching to the "restore" subroutine. This subroutine restores the CIS status bits, the contents of the 2901A registers, and the returning CPC address from the stack. After the CIS has been restored, it returns to the CPC address from which exit was made and continues processing the interrupted microcode.

4.3 DETAILED LOGIC DESCRIPTION
(Reference: CS-M7092, page 6 of 10)

4.3.1 IR Decode

The clock signal for CIS (CIS CLK L) is derived from the processor signal EXT CLK A L. EXT CLK A L is inverted to become CIS CLK H, and this signal is inverted to become CIS CLK L. Cycle time is 180 nsec short cycle and 240 nsec long cycle with 30 nsec off time.

The fetch cycle for the CIS is like the cycle of any other instruction. MPC 10 is latched into the processor instruction register by the deassertion of PROC CLK L, which also asserts LOAD IR L (Figure 4-4). The instruction is fetched at MPC 10 and then decoded in the KD11-Z and the M7092 module of the CISP. The instruction is present on the AMUX lines and since CIS DIS IBUF H (bit 48) is unasserted, the CIS instruction register also has the instruction. The instruction is latched by E69–11 (LOAD IR L) being asserted and E69–12 (CIS CLK L) becoming asserted. CIS INST H (E80-9) is also asserted at this time.

This sequence occurs for all instructions. If a CIS instruction is present, E80–8 (CIS INST L) is asserted, partially enabling E89, the starting address PROM. E89 is completely enabled when CIS CLK L becomes unasserted. At this time, the next CPC address is asserted from the starting address PROM. This CPC address is applied to the control store (M7091) which outputs the starting microinstruction (See Chapter 5 for bit descriptions). This microinstruction contains the 88 bits of information used to direct CIS logic operations. Bits ⟨87:76⟩ contain the next CPC address to execute.

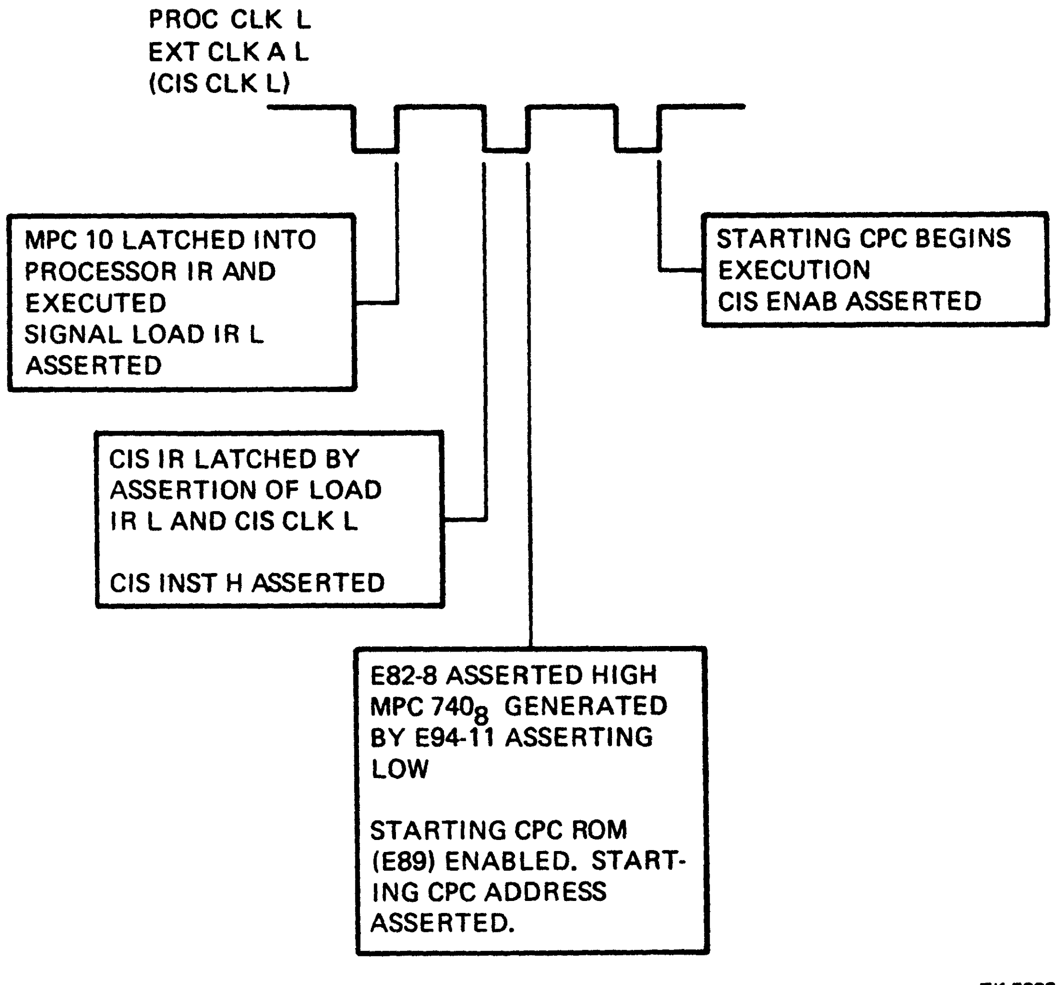

PROC CLK L
EXT CLK A L
(CIS CLK L)
MPC 10 LATCHED INTO PROCESSOR IR AND EXECUTED SIGNAL LOAD IR L ASSERTED
CIS IR LATCHED BY ASSERTION OF LOAD IR L AND CIS CLK L
CIS INST H ASSERTED
E82-8 ASSERTED HIGH MPC 740$_8$ GENERATED BY E94-11 ASSERTING LOW
STARTING CPC ROM (E89) ENABLED. STARTING CPC ADDRESS ASSERTED.
STARTING CPC BEGINS EXECUTION
CIS ENAB ASSERTED
TK-7233

Figure 4-4 MPC Timing

4.3.2 CPC Branching

The next CPC can be modified to branch to a different location if certain conditions exist. For example, a 2901A subtract operation could be executing and a test for carry may be needed. The C-bit would then determine whether the initial CPC, or the CPC "ORed" with bit 01, should be executed next.

Table 4-5 shows which signals can cause branches and the affected CPC bit(s). Branching is caused by "ORing" bits 0, 1 or 2 of the CPC lines.

Table 4-5 CPC Bits Affected by Branching Conditions

Signal Name	CPC Bits Affected
CIS PAGE FAULT H	0
CIS CCZ H	0
CIS SHFT OUT H	0
PFAIL BR PEND H	0
CIS NONZERO A H	0
CIS NONZERO C H	0
CIS IR01 H	0
CIS IR06 H	0
CIS NONZERO A and not CIS NONZERO B	0
CIS CCC H	1
CIS SUB OP H	1
CIS SIGN 2 H	1
CIS DST ADR ODD H	1
CIS SIGN 1 H	1
CIS IR 05 H	1
CIS IR 04 H	1
CIS NONZERO B	1
CIS IR 00 H	1
CIS DST ADR ODD and not CIS SRC1 ADR ODD	1
CIS NONZERO B and not CIS NONZERO A	1
CIS CCN H	2
CIS C/B H	2
CIS SRC 1 ADR ODD H	2
CIS SRC 1 ADR ODD and not DST ADR ODD	2
CIS NONZERO A and CIS NONZERO B	0, 1
CIS DST ADR ODD and CIS SRC1 ADR ODD	1, 2

4.3.3 MPC Addressing

At the same time the starting CPC address is asserted, an MPC of 740_8 is also asserted. This is a result of both E94–13 (CIS INST H) and E94–12 (output of E82–8) being high. The components used to generate the MPC are E110, E111, and E122. Asserting the MPC 740 prevents the base machine from trapping to ten. The base machine itself does not recognize CIS instructions.

4.3.4 Maintenance Switch

Switch S1 selects either the upper or lower part of the starting address PROM (E89). This switch should be off for field use (e.g., when viewing the board from side 1 with the switch at the upper right side of the board, the switch lever should be to the left). The other switch position is used during the manufacturing test.

4.3.5 BCD Operation PROM

IR06 – IR00 connects to the input of the BCD OP PROM (E91) which, during binary coded decimal (BCD) operation, sets up the initial operation of the BCD ALU PROMs (E41, E43). The BCD ALU control signals, called DEC 01 H and DEC 00 H, are obtained from the OP MUX (E73). E73 selects either the initial operation from the BCD OP PROM or a different operation by using OP01H and OP00H (bits 33 and 32 of microword). DEC 01 H and DEC 00 H direct the BCD ALU to one of the operations shown in Table 4-6.

The need for changing operations after the instruction has already been defined, is used, for example, in the divide packed (DIVP) instructions. A DIVP uses successive shift rights and subtracts. The end of the digit string may not be known until one too many subtracts have been completed. In this case an add is needed to restore the string by one digit and will be done by setting DEC 01 and 00 to a value of 00.

Table 4-6 BCD ALU Operations

BCD ALU Control Lines DEC 01	DEC 00	ALU Function
0	0	A + B
0	1	A − B
1	0	B − A
1	1	A × B

4.4 DETAILED BINARY DATA PATH

(Reference: CS-M7092, pages 1, 2, 3, 4 of 10)

The binary data path, as mentioned earlier, centers around the four 2901A bit slices, E44, E45, E46, and E47. (Refer to the 2901A description in Paragraph 4.1 for operational details.)

4.4.1 Direct Data (ALU-In) Multiplexer

The four 2901As, when combined, form seventeen 16-bit registers that are addressed by either the A port (read-only) or the B port (read/write). Data is supplied to the register by the "direct-data-in" lines, or internally from a resulting operation. If the direct data input is used, data can be selected by the ALU-in (direct data) multiplexer from either the internal CIS bus (MBUS) or from the constants circuitry. The ALU-in multiplexer is made up of E24, E14, E39, E40. The signal SELECT ALU IN H (bit 52) to the multiplexer makes the selection.

4.4.2 Constants Generation for 2901A

During CIS instruction, a constant may be needed to count up (or down) the number of bits in a character string, to add two to the PC, or other such operations. The constants PROM (E4) generates these constants by addressing the PROM with CONST SEL S2H – S0H of the control store bits 40–38. The outputs of E4 are applied to E3/E2 (a 74LS298 2:1 multiplexer/latch) which selects either the constants PROM or the MBUS. The output of E3/E2 drives the ALU-in multiplexer or, if ENAB CONST L (bits 27–25) is asserted, also drives the MBUS.

4.4.3 Saving Constants Before Suspension (Interrupt)

A BR request will suspend the CIS instruction. Before the actual suspension occurs, the CIS must clean up and save information on the stack. Constants previously generated must also be saved on the stack. Storing the constants is a two step process.

1. The constant must be enabled to the MBUS by asserting CIS ENAB CONST L. This signal is derived from the CON2 field of the control store, bits ⟨27:25⟩. E13, an octal buffer, then enables the constants to the MBUS.

2. By this time, the CPU will have addressed a stack location. The data on the MBUS must then be pushed onto the stack of the CPU after transmission via the output multiplexer and the AMUX lines. (Paragraph 4.2 gives a more detailed description of suspension protocol.)

4.4.4 Restoring Constants After Suspension (Interrupt)

After completion of a CIS instruction suspension, instruction execution is resumed (from the point of exit) by popping the stack and retrieving the information stored there before suspension. One of these stored pieces of data is the constant.

At this point in the restoration of constants after suspension, the MPC directs the CPU to obtain the information from the stack for transmission to the MBUS via the AMUX lines. The CIS control store then deasserts SEL CONST H and asserts LOAD CONST H. SEL CONST H, being unasserted, selects the 0 input of E3/E2 (the 2:1 multiplexer latch previously referred to), which accepts the MBUS data. LOAD CONST H enables E59 (a 74S00) to latch the MBUS data at the end of the cycle.

4.4.5 2901A Write Operations

The 2901A registers are written to on the trailing edge of the clock only. The upper and lower bytes can be written independently of one another by asserting either CIS SP HIGH WRITE H (Bit 70) and/or CIS SP WRITE H (bit 71) with the trailing edge of the clock. (Figure 5-1 shows the bit fields of the CIS microword and Appendix D gives the meanings of the mnemonics involved.)

The result of the 2901A operation, if selected, can be taken from the 2901A at the Y output (pins 39–36) or can be circulated to another internal register. The output of the 2901A can be enabled by pin 40 (OUT EN) going low if qualified by one of three inputs: FORCE CIS DATA, FREE BUS or DISAB IBUF H (bit 48).

FORCE CIS DATA and FREE BUS, which are generated by the KD11-Z MFM (multifunction) M7096 module, are used to look at the MBUS data. If the 2901A is enabled to the MBUS, the data viewed by the MFM is the 2901A data.

DISAB IBUF H (bit 48) determines whether the 2901A data or the AMUX data is selected as the input to the input multiplexer (E16, E5, E8, E17, E6, E7, E9, and E18).

4.4.6 2901A Shift Operations

The 2901A internal Q-Register and RAM can be connected together to form a 32-bit word that can be shifted left or right. Shift-function electrical connections are shown on page 6 of 10 in the M7092 Print Set; Figure 4-5 shows the results of these connections. SHFT S1 and SHFT S0 are bits 63 and 62, respectively.

4.4.7 Input Multiplexers

The input multiplexer receives:

- The direct output of the 2901A or the direct input AMUX data
- The swapped bytes of the 2901A output or swapped bytes of the AMUX data input
- BCD data in the low byte
- BCD data in the high byte

Swapping is performed in the INPUT MUX (E5, E16, E17, E8, E6, E7, E9 and E18) by asserting the signal SWAP SEL H. That is, while the CIS is fetching data over the AMUX lines, the received high-byte data is swapped; i.e., it appears in the low-byte of the word used by CIS. SWAP SEL H is generated by E71, which selects one of four inputs to determine whether or not to swap. The input signals to E71 are:

Signal	Function
0	No swap
1	Swap
CIS SRC1 ADD ODD H	Swap if SRC 1 ADR was ODD
CIS SRC2 ADD ODD H	Swap if SRC 2 ADR was ODD

These signals are selected by SWAP S1H and SWAP 00H of the control store (bits 50–49).

The output from the INPUT MUX is enabled to the MBUS by asserting INPUT ENAB H (bit 51).

The outputting of data from the MBUS to the AMUX is done by enabling ENAB OBUF H (bit 47) which enables TRI STATE AMUX L and the output multiplexer (E34, E26, E25, E27).

The CPC lines can be enabled onto the AMUX line for viewing on the console terminal by the E/M 1 command. This is accomplished by asserting FORCE CPC L and FREE BUS H from the M7096 multifunction module (MFM) in the CPU.

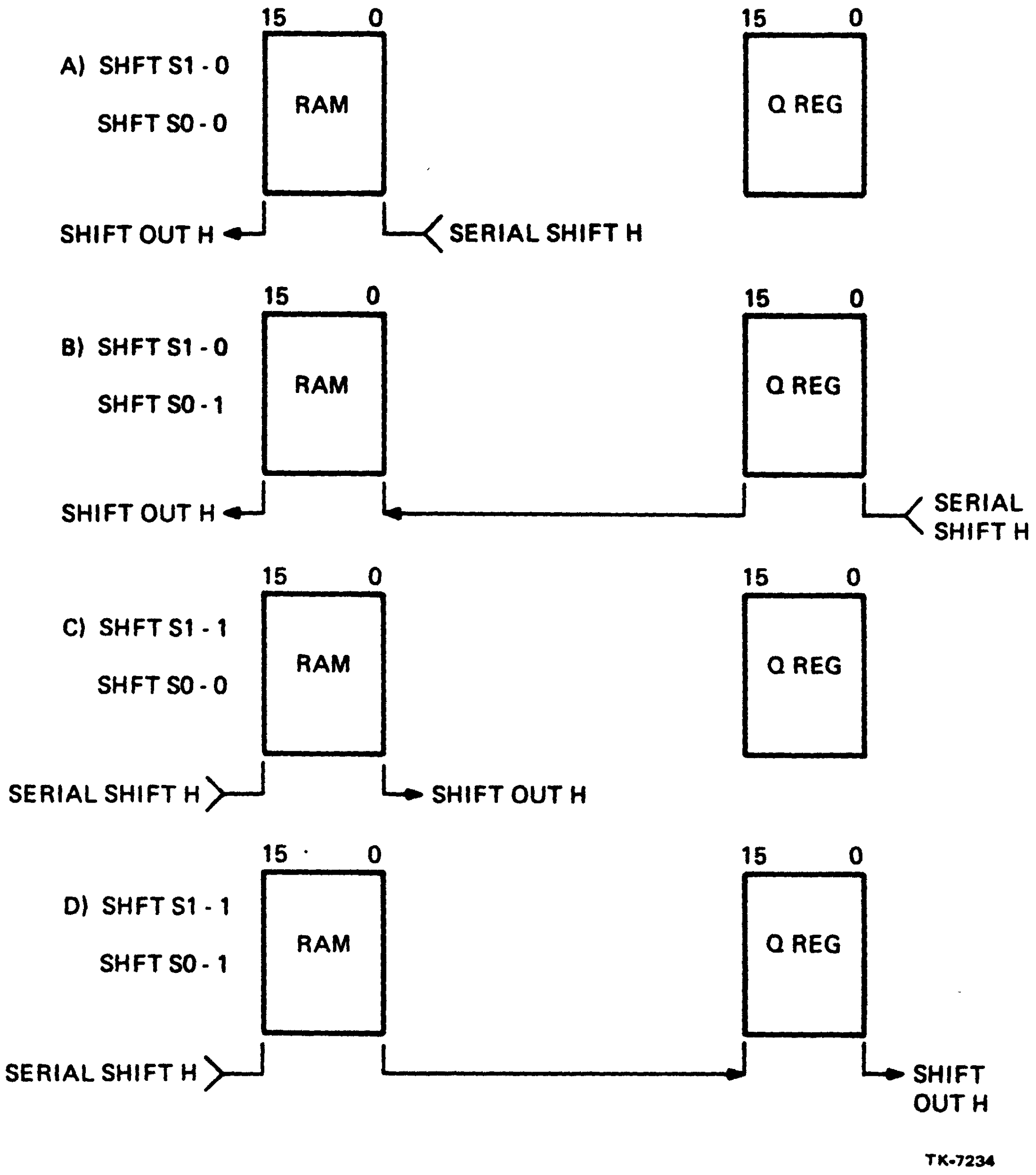


Figure 4-5 2901A Shift Operations

4.5 DETAILED BCD ALU DESCRIPTION

(Reference: CS-M7092, pages 5 and 9 of 10)

The BCD ALU performs its arithmetic by table lookup. A 2-operand add (A+B) applies the two least significant nibbles to a ROM E43 as an address; the output data is an arithmetic result and a carry. If a carry is generated by the addition, it will ripple through to the next arithmetic unit (E41). The arithmetic performed by E41 is identical to that of E43. However, E41 operates on the two most significant nibbles and, if present, a carry from E43. E41 generates the final carry for addition and subtraction.

The ALU can do four operations which are controlled by the DEC 00 and DEC 01 bits (Table 4-6, BCD ALU Operations).

4.5.1 BCD "A" Register/BCD "B" Register

The input operands are obtained by loading the A register (E29, E20) and the B register (E10, E21) with one or two nibbles. The registers are loaded by asserting either LOAD AREG H or LOAD BREG H. The outputs of the registers are latched up at the end of the cycle during the low to high transition of the clock. The output data remains the same until the registers are loaded again or PROC INIT is asserted.

"A" register output is applied directly to the BCD ALU. "B" register output, however, can be shifted left or right, or sent straight through to the BCD ALU. These functions are accomplished in the BCD shift multiplexer (E11, E22, E54, E31, E32) and are selected by BMUX S0 and BMUX S1 (bits 34 and 35). The BCD shift multiplexer can also generate a zero for the BCD ALU.

The shift nibble (E64) stores the "shifted-out" nibble from E11 when BMUX S0 (bit 34) is high. This shifted-out nibble is usually used to hold the sign nibble for the BCD multiplexer before going to the 2901A or MBUS.

4.5.2 BCD Carry

A carry (C/B 3 H) generated from E43 during an add or subtract operation may or may not propogate, depending on the selection made by the 2:1 multiplexer E83. E83 selects either the C/B generated by E43 or, if latched by E102 (a 74S74) in a preceding operation, selects C/BH.

4.5.3 BCD Multiply

The largest addition of two BCD nibbles is 9 + 9. Therefore, an add or subtract operation can only generate a carry of one bit. The answer is 18, where 8 is the low-nibble answer with a carry of one.

A multiply operation can generate a larger carry. That is, the largest multiply can be 9 × 9. The answer is 81, where 1 is the answer with a carry of 8. To obtain the answer, the two operands are still applied to E43 to obtain the low-nibble answer, but the carry is generated by table lookup in a 1K × 4 ROM (E42).

4.5.4 BCD ASCII Encoding

The BCD multiplexer (E51, E49, E76, E53) selects one of the following for input.

- The output of the BCD ALU
- The output of the multiply PROM
- The BLEG output
- A value of 60

These inputs are selected by the signals BCD MUX S3, S2, S1 and S0 (bits 31–28).

The BLEG output, if selected, bypasses the BCD ALU. The value 60 can be tacked onto a nibble used for output in order to produce an ASCII number in the numeric format. The output of the BCD multiplexer is enabled to the input multiplexer (E16, E5, E8, E17, E6, E9, E18) by asserting ENAB SIGN TRAN H (bit 37).

4.5.5 BCD Sign Translation

To effect a necessary translation of the sign nibble in a source or a destination string, the sign nibble is extracted from a source string or is added to a destination string.

Input sign translation is accomplished by the input sign translator (E50). Inputs 14 and 13 to E50 are used to distinguish between packed and zoned formats. BLEG 07 H – BLEG 00 H (which contain the BCD digit and sign) are the other inputs to the translation ROM. The output of the PROM is a BCD number with bit 7 OFF for a positive number or ON for a negative number.

The output sign translator PROM (E74) outputs a BCD number with a sign. This output depends on CIS SIGN H and the numeric or packed format of the instruction. CIS INPUT 12, CIS INPUT 13, and CIS INPUT 14 are used to determine the format of instruction. The BLEG inputs are used if data is to be encoded with the sign.

The output of each sign translator is applied to the sign select multiplexer (E65, E55). One of these signals is selected and then enabled to the BCD lines by the signal CIS ENAB SIGN TRAN H.

4.6 DETAILED LOGIC DESCRIPTION OF STATUS BITS

(Reference: CS-M7092, pages 8, 9, 10 of 10)

The format of the 16-bit CIS status word is shown in Figure 4-6. Bits (3:0) are the condition codes (N, Z, V and C) used by the PDP-11 for branch testing. The status bits (12:04) are used by the CIS for internal branching.

The CIS uses this status word internally and stores it on the stack during suspension. CIS status information other than the condition codes is not available to the user through a register.

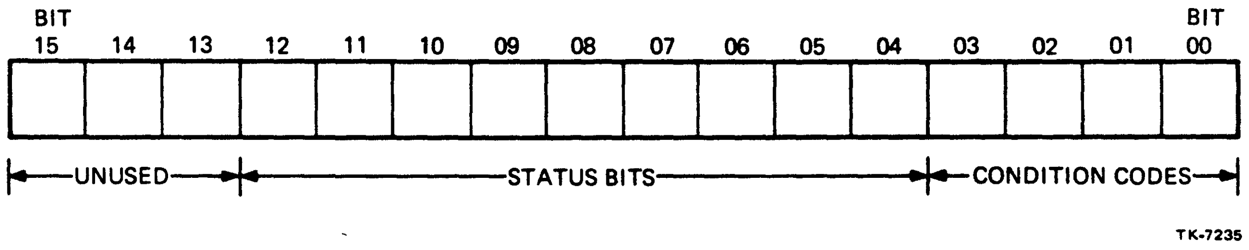


Figure 4-6 CIS Status Word

4.6.1 Status Bits

The status bits (12:04) are set by the result of CIS operations or by conditions of the data string. A CIS operation could set the following bits.

- Address odd conditions (bits 12:10)
- Sign (status bits 9, 8)
- Zero condition (status bits 7:5)
- Carry/borrow (status bit 4)

4.6.2 Nonzero Conditions

(Reference: CS-M7092, page 5 of 10)

The zero conditions are set or cleared after a BCD ALU operation. Two signals, CIS BCD 3:0=0 H and CIS BCD 7:4=0 H, are used to generate the three zero condition bits. These signals monitor the low- and high-nibble respectively, of the BCD arithmetic ROMs. The negated state of the zero condition bits is used for the status indication. The signal names are therefore referred to as NONZERO and have three versions:

1. NONZERO A (bit 7)
2. NONZERO B (bit 6)
3. NONZERO C (bit 5)

All three status bits may be used, if the status of the source 1, the source 2, and the destination need to be known. This can occur in three address arithmetic when using an ADD, SUB, MULP or DIVP.

Each of the NONZERO flip-flops is independently enabled. These signals originate from the M7091 control store and enable the signal CIS CLK H, which latches the NONZERO flip-flops 30 nsec before the end of the CIS cycle. The enabling signal names are:

1. ENAB NONZERO A H
2. ENAB NONZERO B H
3. ENAB NONZERO C H

These signals are derived from the CON4 field of the control store, bits (20:16).

The nonzero signals, which are latched by 74S74 flip-flops, stay set until a CLEAR NONZERO H signal is received from the control store or a zero condition is latched. A CLR NONZERO or PROC INIT signal clears all NONZERO flip-flops.

4.6.3 Carry/Borrow
The carry/borrow (C/B) status bit (bit 4) is set or cleared by a carry-out during a BCD operation. The carry bit is latched up by E102 (a 74S74) if the enable signal ENAB C/B H is present. ENAB C/B H is the ENCB field (bit 0) of the control store. A C/B may also be forced by the control store if CIS FORCE C/B H is asserted. This signal is derived from the CON4 field of the control store (bits 20:16).

The carry/borrow status bit can be selected from either the high or low nibble. That is C/B OUT H or C/B 3 H from the BCD ALU will be inputted to the C/B latch. CIS LNIB SEL H (low-nibble select) selects either C/B H or C/B 3 H and is derived from the CON4 field (bits 20:16). The selection is done by 4:1 multiplexer E96.

4.6.4 Sign Bits
Two sign bits are available to store the sign for two source operands. These bits are latched for use in setting the condition codes during character string instructions, or for CPC address branching.

The two sign bits, CIS SIGN 1 H (status bit 8) and CIS SIGN 2 H (status bit 9), are derived from the signal CIS DAT SIGN H at E70 pin 9. CIS DAT SIGN H is produced from one of the following signals, depending upon the data type of the instruction.

CIS DAT SIGN H is set during character instructions, if data bit 15 or 7 on the MBUS is set. This is the sign bit of either the high byte or the low byte.

CIS DAT SIGN H is set during long integer instructions, if bit 15 (the sign bit) of the MBUS is set.

CIS DAT SIGN H is set during zoned string instructions if bit 6 of the MBUS data is set. The state of bit 6 represents the difference between a positive or negative number in the zoned format. A byte with a positive signed number is represented as 0011 xxxx and a byte with a negative signed number is represented as 0111 xxxx, where xxxx is a valid BCD number.

CIS DAT SIGN H is set during packed data instructions if bit 0 of the MBUS data is set. The state of bit 0 represents the difference between a positive or negative signed number in the packed format. A positive number with sign is represented as xxxx 1100 while a negative number with sign is xxxx 1101, where xxxx is a valid BCD number. Notice that the state of bit 0 in the nibble differentiates the positive from the negative numbers.

DAT TYPE 01 H and DAT TYPE 00 H are the signals that select the correct bit for CIS DAT SIGN H. Table 4-7 shows their functions.

The SIGN 1 and SIGN 2 flip-flops (enabled by ENAB SIGN 1 H and ENAB SIGN 2 H respectively) are derived from the CON4 field of the control store (bits 20:16). The SIGN 1 and SIGN 2 latches are cleared by PROC INIT or by loading a positive sign bit.

Table 4-7 Sign 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

4.6.5 Address Odd Conditions

Since CIS data strings can start or stop on odd address boundaries, the determination of whether writing is to a word or byte, is made by the CIS. Three signals are available for determining whether the sources and/or the destination addresses are odd:

CIS SRC 1 ADR ODD H
CIS SRC 2 ADR ODD H
CIS DST ADR ODD H

Each of these signals monitors the MBUS 00 signal. An odd address condition is set if the address loaded to the MBUS has bit 0 set. The microcode will test these three signals and branch if at least one of the indicated conditions is set.

4.6.6 Status Bit Operation With BR Interrupt Pending

A test is made at specific points in the microcode for the presence of a BR interrupt request. A successful test (i.e., a BR interrupt is detected) will branch the microcode to a "save state subroutine" that stores essential information on the stack. One such piece of information is the CIS status word.

The status word is first enabled to the MBUS, then to the AMUX, and finally to the stack. The MBUS receives the status bits via buffers E12 (status bits 7:0), and E92 (status bits 15:8). The signal CIS ENAB CISS L, which enables the status information to the MBUS, is derived from the CON2 field (bits 27:25) of the control store.

4.6.7 Return From Interrupt

After an interrupt has been serviced and control is returned to the CIS, the CIS must continue from where it left off. The stack is popped twice after returning from the interrupt, first to load the PC and then to load the PSW. The instruction then continues as though suspension had not occurred. The PSW is examined by the CIS mirocode and, if PSW bit 8 is set (indicating that a CIS instruction was suspended), a restore subroutine is called to pop previous information off the stack.

One of the items popped from the stack is the status word, which is placed on the AMUX lines and enabled to the MBUS via the input multiplexer (E16, E5, E8, E17, E6, E7, E9, E12). The MBUS connects to the multiplexers which drive the status latches. The multiplexers used to restore status information are E62, E30 (74S153), E93 (74S157), and E96 (74S153). The MBUS signals (status bits) are enabled to the latches by CIS SEL CISS L which is derived from the CON2 field of the control store (bits 27:25). The status word and all pertinent information is restored from the stack thus allowing continuation of the interrupted instruction.

4.6.8 Categorizing Instructions To Form N, Z, V, C Bits

(Reference drawing: CS-M7092, page 8 of 10)

The condition codes are formed by first categorizing similar instruction (Table 4-8). Two general categorizing groups exist: character string instructions and arithmetic instructions. Within each group are subgroups. Specifically, character string instructions are divided into two groups and arithmetic instructions into eight groups. Each subgroup comprises all instructions that output similar condition codes. Functionally, this grouping takes place in E100 which is a 256 × 8 PROM. This PROM also outputs two signals indicating the format of the data type. These signals (DAT TYPE 00 and DAT TYPE 01) are only used to form the output sign for character instructions.

The categorizing logic is divided into two sections: character string condition codes and arithmetic condition codes. Condition codes for arithmetic instructions are formed by using the four categorizing signals from E100 and some status bits. The character string condition codes are derived from the 2901A status bits.

Table 4-8 Instruction Categories

Categorizing Instruction	High CC SEL L	CC Code 02H	CC Code 01H	CC Code 00H
MOVC,MOVRC,MOVTC,CMPC	0	0	0	1
MATC,LOCC,SKPC,SCANC,SPANC,L3Dx,L2Dx	0	0	0	0
ADDy	1	0	0	1
SUBy	1	0	1	0
DIVP	1	0	1	1
MULP	1	1	0	0
CMPy	1	1	0	1
ASHy	1	1	1	0
CVTLy	1	1	1	1
CVTyL	1	0	0	0

CVTLy	Data type is long integer.
CVTNP	Data type is ZONED.
CVTPN	Data type is PACKED.
CVTyL	Data type is ZONED or PACKED.

where x = any number in the range 0-7
y = P or N

4.6.9 Arithmetic Condition Codes
The status bits and the categorized group of the instructions, set the arithmetic condition codes (N, Z, V, C bits). The status bits are formed by the results of the BCD ALU and are as follows:

CIS C/B H
CIS NONZERO A H
CIS NONZERO B H
CIS NONZERO C H
CIS SIGN 2
CIS SIGN 1

These signals, together with the categorizing ROM (E100) output, address the decimal condition code ROM (E79), which then outputs the N, Z, V, C bits and a sign bit. The sign bit will be set under the following sign 1 and sign 2 settings:

Instruction	**Condition**
ADDx, SUBx	Sign 2
DIVP, MULP	Sign 1 XOR Sign 2
CPMx	Sign not set
ASHx, CVTxx	Sign 1

The condition code settings for each instruction are given in Table 4-9.

4.6.10 Condition Code Output
The condition codes are selected by two dual 4:1 multiplexers (E62, E30). The inputs to these multiplexers are either the decimal CC decode ROM, character condition codes, or the MBUS.

After the correct input is selected, the output of the multiplexer is stored in the condition code latch (E19). The data can then be fed to the PSW via the MBUS and AMUX lines of the CIS.

4.6.11 Character String Condition Codes
The character string condition codes are set by monitoring the status information on the 2901A ALUs.

The 2901As F = 0 output of either high byte or low byte can set the Z bit. The signal names are CIS ALU 15:8=0 H, and CIS ALU 7:0=0 H. CIS ALU 15:8=0 H indicates that the high byte is zero; CIS ALU 7:0=0 H indicates that the low byte is zero.

The carry bit (C) indicates a carry-out of the 2901A, and either the high byte (CIS ALU COUT H) or low byte (CIS ALU COUT 7) can be selected to obtain the signal. CIS ALU COUT 7 H is generated by the carry-lookahead chip, E48. CIS ALU COUT 7 H is generated as a separate output by the most significant nibble of the 2901A, E47.

The negative bit (N) is set if the sign bit of either the low or high byte is set. Two signals (ALU 15 H and ALU 07 H) from the 2901A can set the N-bit.

The overflow bit (V) is used only during a MOVC, MOVRC, MOVTC or CMPC instruction. During other character string instructions the V-bit is zero. The V-bit is set by the simple Boolean expression found in the Condition Code Setting Table 4-9, V column.

Table 4-9 Condition Code Settings

Instruction	N
MOVC, MOVRC MOVTC, CMPC	ALU7 or ALU15
LOCC, SKPC, SCANC, SPANC, MATCHC	ALU15
ADDN, ADDP	SIGN2 · NONZEROA
SUBN, SUBP	SIGN2 · NONZEROA
DIVP	(SIGN1 ∀ SIGN2) · NONZEROA
MULP	(SIGN1 ∀ SIGN2) · NONZEROA
CMPN, CMPP	SIGN1 · $\overline{\text{SIGN2}}$ [C/B · (NONZEROB · C/B) · NONZEROA] + (C/B · $\overline{\text{SIGN1}}$) + (SIGN1 · SIGN2 · NONZEROA · $\overline{\text{C/B}}$)
ASHN, ASHP	SIGN1 · NONZEROA
CVTLN, CVTLP CVTPN, CVTNP	SIGN1 · NONZEROA
CVTNL, CVTPL	SIGN1 · NONZEROA

∀ = XOR

Z	V	C
ALU<7:0> =0 or ALU<15:0> =0	$(\text{SIGN1} \forall \text{SIGN2}) \cdot$ $[\overline{\text{SIGN2} \forall (\text{ALU15}+\text{ALU07})}]$	$\overline{\text{ALU COUT}}$
ALU<15:0> =0	0	0
$\overline{\text{NONZEROA}}$	C/B + NONZEROB	0
$\overline{\text{NONZEROA}}$	C/B + NONZEROB	0
$\overline{\text{NONZEROA}}$	C/B + NONZEROB + $\overline{\text{NONZEROC}}$	$\overline{\text{NONZEROC}}$
$\overline{\text{NONZEROA}}$	C/B + NONZEROB	0
$<[(\text{SIGN1} \forall \text{SIGN2}) \cdot \overline{\text{NONZEROA}} \cdot \overline{\text{NONZEROB}} \cdot \overline{\text{C/B}}] +$ $[\overline{(\text{SIGN1} \forall \text{SIGN2})} \cdot (\overline{\text{NONZEROA}} + \overline{\text{C/B}})]>$	0	0
$\overline{\text{NONZEROA}}$	C/B + NONZEROB	0
$\overline{\text{NONZEROA}}$	C/B + NONZEROB	0
$\overline{\text{NONZEROA}}$	C/B + NONZEROB	SIGN2 · NONZEROC

$\forall$ = XOR

CHAPTER 5
MICROCODE

5.1 INTRODUCTION

The KE44-A microcode (control store) consists of 1,000 88-bit words. Each word of the microcode controls an operation or a set of operations within the KE44-A, as well as the selection of the next microword. Initial microword selection, however, is controlled by the op code of the CIS instruction to be performed. (Refer to Appendix A for a description of these instructions.) When a valid CIS op code (076 nnn) is received, a microword (specified by the decode of the op code) is addressed, and control of KD11-Z operation transfers to the KE44-A. The series of operations specified by the microwords contained in the addressed op code routine is then performed. This sequence also includes subroutines called for by the addressed routine. Upon completion of the tasks called for by the instructions, the KE44-A is returned to the idle state address (0000) and control of system operation is returned to the KD11-Z. In this idle state, the KE44-A monitors KD11-Z operation in order to detect any valid CIS op code transmitted on the AMUX ⟨15:00⟩ lines.

5.1.1 Design Guideline

DEC STD 168, PDP-11 Extended Instructions is the design guideline for the CIS microcode.

5.1.2 Microcode Listing

The contents of the control store are described in a computer listing of definitions used in the instructions. These definitions include a detailed description of each microword broken down by operational areas (fields), and a definition of the macros (Appendix A) used in the microword instructions.

The microword instructions are listed by routine or subroutine and, in general, appear in their sequence of occurrence within that category. The listing begins with general (nonroutine-related) microwords, for example, "CIS idle state".

Each microword address is accompanied in the microcode listing by a description of the operations to be performed when the microword is implemented. Each description is followed by a listing of the values for each field (given under the number representing the location of the least significant digit (LSD) for that field). Each entry ends with a to/from listing showing the words that can be entered from a given word as well as the words from which the given word stemmed.

5.2 THE MICROWORD

The microword contains 88 bits ⟨87:00⟩. These bits are divided into groups called fields and subfields, which control operations internal to the KE44-A and addresses to the KD11-Z microstore. Figure 5-1 shows the microword field map.

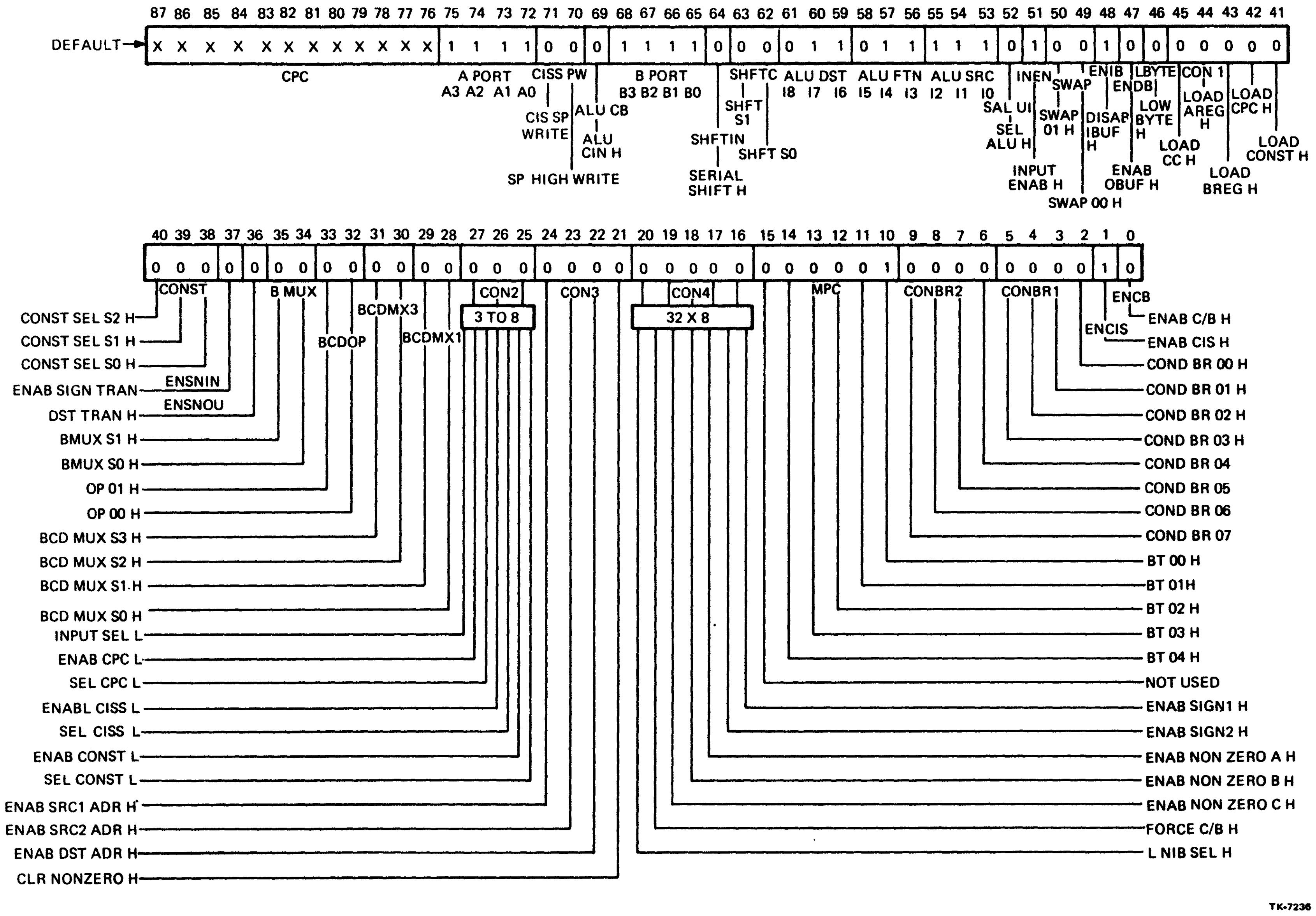


Figure 5-1 CIS Microword Field Map

5.2.1 CPC Field ⟨87:76⟩
The CISP program counter (CPC) field is the next microword address pointer (0000_8 through 1777_8). The CPC field output can be modified during KE44-A operation by the results of tests called for by the CONBR1 and CONBR2 fields and/or condition codes.

5.2.2 APORT Field ⟨75:72⟩
The APORT field determines which of the 16 working registers in the 2901A data processor of the binary data path are to be read by the APORT. The default for this field (APORT = 17) is "register 17".

5.2.3 CISSPW Field ⟨71:70⟩
The CISP scratch pad write field (CISSPW) enables the writing of a data byte or word to the 2901A ALU registers via the BPORT. The default for this field (CISSPW = 0) is "disable writing to the registers".

5.2.4 ALUCB Bit ⟨69⟩
The ALU carry/borrow bit (ALUCB) controls the carry/borrow operation for the binary path ALU. The default for this bit (ALUCB = 1) is "no carry/borrow input".

5.2.5 BPORT Field ⟨68:65⟩
The BPORT field determines which of the 16 registers in the 2901A binary data path are to be written to and/or read from. The default for this field (BPORT = 17) is "register 17".

5.2.6 SHFTIN Bit ⟨64⟩
The shifted-in bit (SHFTIN) controls the value (1 or 0) of data shifted in. The default for this field (SHFTIN = 0) is "shift in zero".

5.2.7 SHFTC Field ⟨63:62⟩
The shift control field (SHFTC) controls the direction of shift for data being loaded into the binary path RAM and/or Q-register. The default for this field (SHFTC = 0) is "left shift one bit in RAM register if enabled by ALUDST ⟨61:59⟩".

5.2.8 ALUDST Field ⟨61:59⟩
The ALU destination field (ALUDST) controls the form of the 2901A ALU output in the binary data path (i.e., whether the output is RAM data or calculated output) and the data input to the BPORT and the Q-register. The default for this field (ALUDST = 3) is "read the calculated output to the Y output and back to the BPORT".

5.2.9 ALUFTN Field ⟨58:56⟩
The ALU function field (ALUFTN) controls the arithmetic/logical operation to be performed by the ALU of the 2901A data processor in the binary data path. The default for this field (ALUFTN = 3) is "selects a logical OR operation of the ALU input". ALU inputs are selected by the ALU SRC field.

5.2.10 ALUSRC Field ⟨55:53⟩
The ALU source field (ALUSRC) controls the selection of data sources as inputs to the binary path ALU. The default for this field (ALUSRC = 7) is "selects the direct data for the R input and zero for the S input to the ALU".

5.2.11 SALUI Bit ⟨52⟩
The select ALU input bit (SALUI) controls the input multiplexer selection for the binary path ALU (2901A) direct path (D) input. The default for this bit (SALUI = 0) is "transfer the contents of the MBUS to the D input".

5.2.12 INEN Bit ⟨51⟩
The input enable bit (INEN) is the enable/inhibit control for the tri-state output of the eight input multiplexers. The default for this bit (INEN = 0) is "inhibit output". When the input multiplexer is enabled, data is put onto the MBUS.

5.2.13 SWAP Field ⟨50:49⟩
The SWAP field controls the swapping of bytes in a word or in a data string. The swapping operation is performed in the reading of data from the input multiplexer. The default for this field (SWAP = 0) is "inhibit the swap of byte data".

5.2.14 ENIB Bit ⟨48⟩
The enable input buffer bit (ENIB) is the enable/inhibit control for the tri-state buffers, the AMUX input line and the ALU Y output lines. The default for this field (ENIB = 1) is "enable ALU Y outputs" (AMUX inputs inhibited).

5.2.15 ENOB Bit ⟨47⟩
The enable output buffer bit (ENOB) is the enable/inhibit control for the tri-state output of the output multiplexer. When set, this bit enables data from the MBUS to the AMUX lines. The default for this bit (ENOB = 0) is "inhibit output multiplexer output".

5.2.16 LBYTE BIT ⟨46⟩
The low byte enable bit (LBYTE) controls the location (high or low byte) of the condition codes. The default for this bit (LBYTE = 0) is "load the condition codes into the high byte".

5.2.17 CON1 Field ⟨45:41⟩
The control 1 field (CON1) controls the selection of internal CIS data for loading onto the MBUS. The default for this field (CON1 = 0) is "inhibit loading data to MBUS".

5.2.18 CONST Field ⟨40:38⟩
The constant field (CONST) selects which of the eight constants from the constants ROM is enabled as the input to the constant multiplexer. The default for this field (CONST = 0) is "enable the constant 7".

5.2.19 ENSNIN Bit ⟨37⟩
The enable sign input bit (ENSNIN) enables/inhibits the translation of the input sign data. The default for this bit (ENSNIN = 0) is "inhibit sign translation".

5.2.20 ENSNOU Bit ⟨36⟩
The enable sign output bit (ENSNOU) enables/inhibits the translation of the output sign data. The default for this bit (ENSNOU = 0) is "inhibit sign translation".

5.2.21 BMUX Field ⟨35:34⟩
The B multiplexer field (BMUX) controls the selection of data to be read from the BCD shift multiplexer and from the shift nibble register. The default for this field (BMUX = 0) is "read the contents of the B register to the BCD shift multiplexer unshifted".

5.2.22 BCDOP Field ⟨33:32⟩
The BCD operation field (BCDOP) controls the operations to be performed by the BCD ALU. The default for this field (BCDOP) is "use the decode of the BCD operation PROM to control the BCD ALU".

5.2.23 BCDMX3 Field ⟨31:30⟩
The BCD multiplexer 3 field (BCDMX3) controls the selection of the output of the high-nibble multiplexer data to the input multiplexer. The default for this field (BCDMX3 = 0) is "read the output of the BCD ALU unchanged".

5.2.24 BCDMX1 Field ⟨29:28⟩
The BCD multiplexer 1 field (BCDMX1) controls the selection of the low-nibble multiplexer data to the input multiplexer. The default for this field (BCDMX = 0) is "read the BCD ALU output unchanged".

5.2.25 CON2 Field ⟨27:25⟩
The control 2 field (CON2) comprises the enabling signals for control of the data input to the MBUS. This field works in conjunction with CON1. The default for this field (CON2 = 0) is "inhibit all enabling signals (no data to the MBUS)". The three bits of this field are converted to eight signal lines on the M7091 control store module.

5.2.26 CON3 Field ⟨24:21⟩
The control 3 field (CON3) comprises the enabling signals for the latching of odd address conditions in the source and destination. The default for this field (CON3 = 0) is "disable all enabling".

5.2.27 CON4 Field ⟨20:16⟩
The control 4 field (CON4) is a series of enabling signals for CIS operations. The five bits of this field are decoded by the M7091 control store module into seven control signals. The default for this field (CON4 = 0) is "disable all enabling".

5.2.28 MPC Field ⟨15:10⟩
The microprogram counter field (MPC) comprises control signals that go to the KD11-Z. These signals are read from MPC decode to the MPC line ⟨8:0⟩. The resulting decode of the MPC field is in the range from 740_8 to 776_8. The default for this field (MPC = 1) is 741.

5.2.29 CONBR2 Field ⟨9:6⟩
The conditional branch 2 field (CONBR2) is used in conjunction with the condition codes, to generate branch conditions in the CPC\2:0÷ field. The default for this field (CONBR2 = 0) is "inhibit all conditioned branches".

5.2.30 CONBR1 Field ⟨5:2⟩
The conditional branch 1 field (CONBR1) is used by the FPLA to generate branch signals for CPC ⟨7:0⟩. The default for this field (CONBR1 = 0) is "inhibit all branch conditions".

5.2.31 ENCIS Bit ⟨1⟩
The enable CIS bit (ENCIS) controls the CIS operational mode. When ENCIS = 0, the KD11-Z is in control. When ENCIS = 1, the KE44-A controls the operation; i.e., a CIS instruction has been decoded.

5.2.32 ENCB Bit ⟨0⟩
This bit controls the loading of the carry/borrow bit to the MBUS and is used in BCD operations.

5.3 READING THE MICROCODE
Reading the microcode listing involves a series of steps. These steps vary according to the contents of the microword and the familiarity of the user with the appropriate tools. The tools available include the field definitions, the sets of macro-definitions, the KD11-Z operations, and the microword listing with its descriptions.

5.3.1 The Field Definitions

Figure 5-2 is a sample page of microcode field definitions. Note that each field is defined by: 1) a symbol, 2) its position and range in the microword, and 3) its default value:

SYMBOL/= ⟨n:nw⟩, default = m

```
Microflow 0A(00) 12:41:08 26-Mar-1981 CIS004.FLW 44-CIS
CIS1.MCR

                .RTOL                   ;THE DEFINITION OF THE WORD IS RIGHT TO LEFT
                .TOC  "44 CIS MICROWORD DEFINITIONH"



        CPC/=<87:76>,.NEXTADDRESS       ;CONTAINS ADDR OF NEXT MICROWORD
                        BK2=0           ;FIRST 2K OF ROMS
                        BK4=1           ;SECOND 2K OF ROMS
        APORT/=<75:72>,.DEFAULT=17      ;WHICH REG TO BE READ BY APORT
                        K0=00
                        K1=01
                        K2=02
                        K3=03
                        K4=04
                        K5=05
                        K6=06
                        K7=07
                        K10=10
                        K11=11
                        K12=12
                        K13=13
                        K14=14
                        K15=15
                        K16=16
                        K17=17
        CISSPW/=<71:70>,.DEFAULT=0      ;ENABLES 2901 WRITE OPTION
                        SPNW=0          ;DISABLE WRITE INTO RAM
                        HBYTE=1         ;WRITE HIGH BYTE AND LOW BYTE
                        LBYTE=2         ;WRITE ONLY LOW BYTE
                        SPW=3           ;WRITE INTO RAM
        ALUCB/=<69>,.DEFAULT=1          ;CARRY BIT IF CONTROL STORE
                        NOCAR=1         ;NO CARRY/BORROW IN
                        YESCAR=0        ;CARRY IN OF 1
        BPORT/=<68:65>,.DEFAULT=17      ;CONTROLS REG FOR BPORT READ/WRITE
                        K0=00
                        K1=01
                        K2=02
                        K3=03
                        K4=04
                        K5=05
                        K6=06
                        K7=07
                        K10=10
                        K11=11
                        K12=12
                        K13=13
                        K14=14
                        K15=15
                        K16=16
                        K17=17
        SHFTIN/=<64>,.DEFAULT=0         ;BIT TO BE SHIFTED IN BY SHIFT COMMANDS
                        ZERO=0          ;SHIFT IN A ZERO
                        ONE=1           ;SHIFT IN A 1
        SHFTC/=<63:62>,.DEFAULT=0       ;CONTROLS SHIFTS IF A SHIFT OCCURS
                        LFT16=0         ;LEFT SHIFT OF ONE BIT USING 16 BIT REG
```

Figure 5-2 Sample Page of Microcode Field Definitions

The symbol is a signal name or mnemonic followed by a slash (/) followed by an equal sign (=).

The bracketed numbers(s) (⟨n:nw⟩) show the location and range of the field in the 88-bit microword. N is the MSD and NW is the LSD of the field with the microword. The default is the value (m) to which the field is set if no value is given for that field in a given microword. This entry is followed by a descriptive statement of the field.

The field definition entry is followed by a series of entries defining the resulting action or value for the bit combinations within the field.

5.3.2 The Microinstruction

Figure 5-3 is a sample page of microinstructions. Each microinstruction in the listing is an entry in the control store. The microwords are grouped by routine or subroutine, and each group is identified by a table of content (TOC) entry. The comment is repeated at the top of each page containing a given routine or subroutine.

Each microword entry begins with an identifying label. The label, an identifying symbol for the microword, is used by the other microwords to call (go to) that address. If more than one label is given, any of these can be used as a call. Each label is followed by a colon, e.g., 0000: and SERV: (Figure 5-3).

A label may be followed by a descriptive statement appearing on the same line. The statement identifies a specific detail (i.e., why it was called or what it will do) related to that word. The descriptive statement is separated from the label by two semicolons.

The line following the label contains macrostatements identifying the operations performed as a result of asserting the microword. Each macrostatement line may also contain a short descriptive statement about macrooperation. The descriptive statement is separated from the macro by a semicolon. Macro entries are followed by the address of the microword, a six digit octal number in brackets, e.g., [000000] (Figure 5-3).

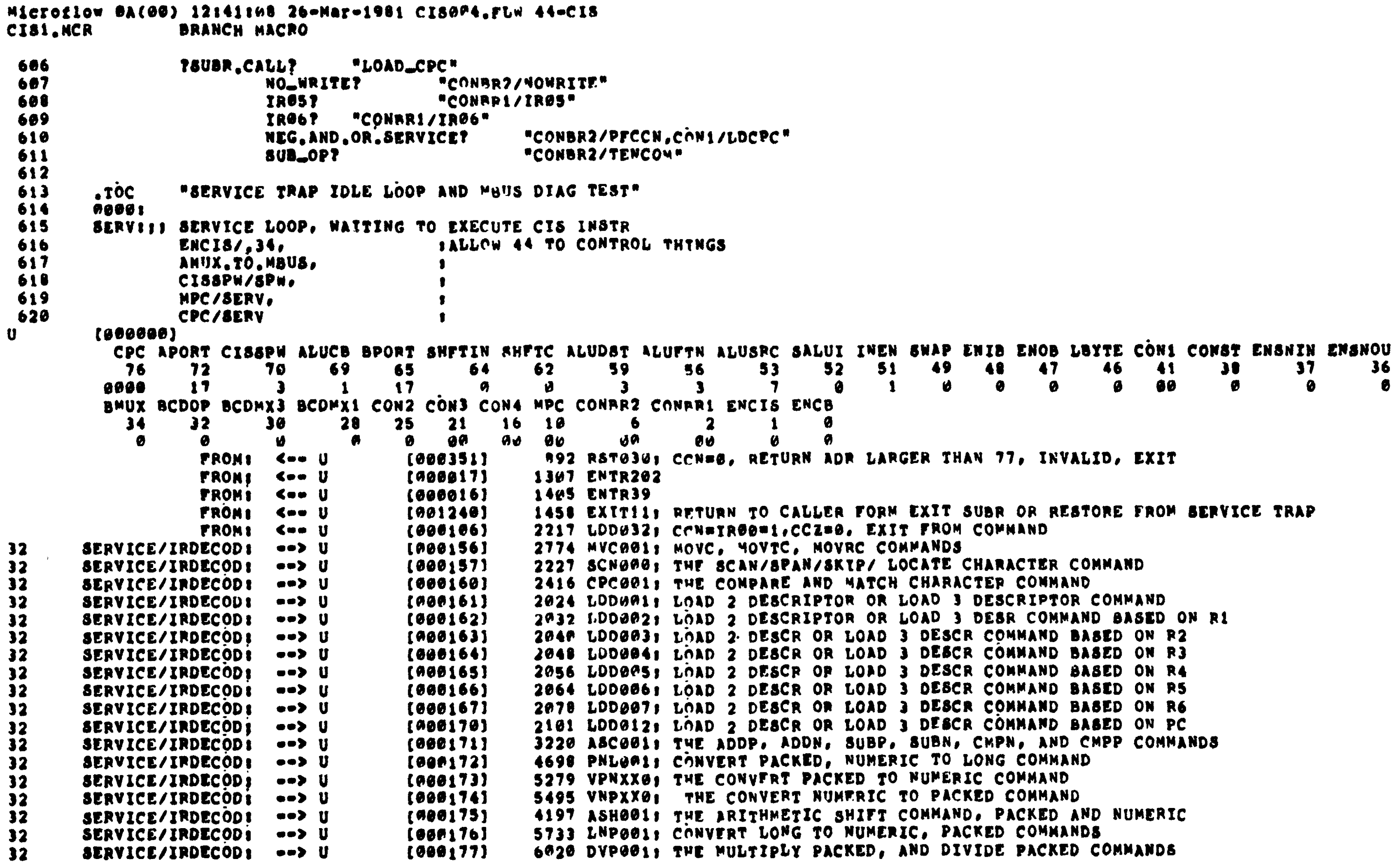
The microword address is followed by two sets of 3 lines; each set gives, for example, the field name (CPC), the LSD of the field (76), and the bit value for that field (0000).

For the idle state (address 0000) the next address is 0000. The KE44-A repeats this operation until a valid CIS op code (076 nnn) is received.

The contents of the microword entry is followed by a list of to/from entries (e.g., from: ← U [00035]). The value 892, which follows this number, is the cross-reference (CREF) number for microword 000351. The "from" entries show the origin of this microword. The "jump" form (not shown in Figure 5-3), is for the next address only. When a branch condition exists, all possible addresses and the condition for selection are given. In the idle state the next word address is determined by the decode of the nnn part of the 076 nnn op code.

5.3.3 Reading the Macrodefinitions

Figure 5-4 is a sample page of macrodefinitions. The macrodefinitions are grouped by functions and each group is identified by a table of contents (TOC) entry which defines the function of the macros. Under each TOC entry are two columns. The first column is the macrocode listing; the second column is a definition of the macro. In some cases, a macro may have already been defined at a previous point in the listing, e.g., TEMP-2901 (Figure 5-4).



```
Microflow 0A(00) 12:41:08 26-Mar-1981 CIS0P4.FLW 44-CIS
CIS1.MCR          BRANCH MACRO

 606           ?SUBR.CALL?       "LOAD_CPC"
 607                   NO_WRITE?          "CONBR2/NOWRITE"
 608                   IR05?              "CONBR1/IR05"
 609                   IR06?   "CONBR1/IR06"
 610                   NEG.AND.OR.SERVICE?     "CONBR2/PFCCN,CON1/LDCPC"
 611                   SUB_OP?                 "CONBR2/TENCOM"
 612
 613       .TOC    "SERVICE TRAP IDLE LOOP AND MBUS DIAG TEST"
 614       0000:
 615       SERV:;; SERVICE LOOP, WAITING TO EXECUTE CIS INSTR
 616               ENCIS/,34,                  ;ALLOW 44 TO CONTROL THINGS
 617               AMUX.TO.MBUS,               ;
 618               CISSPW/SPW,                 ;
 619               MPC/SERV,                   ;
 620               CPC/SERV                    ;
U          [000000]
            CPC APORT CISSPW ALUCB BPORT SHFTIN SHFTC ALUDST ALUFTN ALUSRC SALUI INEN SWAP ENIB ENOB LBYTE CON1 CONST ENSNIN ENSNOU
             76    72     70    69    65     64    62     59     56     53    52   51   49   48   47    46   41    38     37     36
           0000    17      3     1    17      0     0      3      3      7     0    1    0    0    0     0   00     0      0      0
            BMUX BCDOP BCDMX3 BCDMX1 CON2 CON3 CON4 MPC CONBR2 CONBR1 ENCIS ENCB
              34    32     30     28   25   21   16  10      6      2     1    0
               0     0      0      0    0   00   00  00     00     00     0    0
                     FROM:   <-- U        [000351]       892 RST030; CCN=0, RETURN ADR LARGER THAN 77, INVALID, EXIT
                     FROM:   <-- U        [000017]      1307 ENTR202
                     FROM:   <-- U        [000016]      1405 ENTR39
                     FROM:   <-- U        [001240]      1458 EXIT11; RETURN TO CALLER FORM EXIT SUBR OR RESTORE FROM SERVICE TRAP
                     FROM:   <-- U        [000106]      2217 LDD032; CCN=IR00=1,CCZ=0, EXIT FROM COMMAND
32      SERVICE/IRDECOD:     --> U        [000156]      2774 MVC001; MOVC, MOVTC, MOVRC COMMANDS
32      SERVICE/IRDECOD:     --> U        [000157]      2227 SCN000; THE SCAN/SPAN/SKIP/ LOCATE CHARACTER COMMAND
32      SERVICE/IRDECOD:     --> U        [000160]      2416 CPC001; THE COMPARE AND MATCH CHARACTER COMMAND
32      SERVICE/IRDECOD:     --> U        [000161]      2024 LDD001; LOAD 2 DESCRIPTOR OR LOAD 3 DESCRIPTOR COMMAND
32      SERVICE/IRDECOD:     --> U        [000162]      2032 LDD002; LOAD 2 DESCRIPTOR OR LOAD 3 DESR COMMAND BASED ON R1
32      SERVICE/IRDECOD:     --> U        [000163]      2040 LDD003; LOAD 2 DESCR OR LOAD 3 DESCR COMMAND BASED ON R2
32      SERVICE/IRDECOD:     --> U        [000164]      2048 LDD004; LOAD 2 DESCR OR LOAD 3 DESCR COMMAND BASED ON R3
32      SERVICE/IRDECOD:     --> U        [000165]      2056 LDD005; LOAD 2 DESCR OR LOAD 3 DESCR COMMAND BASED ON R4
32      SERVICE/IRDECOD:     --> U        [000166]      2064 LDD006; LOAD 2 DESCR OR LOAD 3 DESCR COMMAND BASED ON R5
32      SERVICE/IRDECOD:     --> U        [000167]      2078 LDD007; LOAD 2 DESCR OR LOAD 3 DESCR COMMAND BASED ON R6
32      SERVICE/IRDECOD:     --> U        [000170]      2101 LDD012; LOAD 2 DESCR OR LOAD 3 DESCR COMMAND BASED ON PC
32      SERVICE/IRDECOD:     --> U        [000171]      3220 ASC001; THE ADDP, ADDN, SUBP, SUBN, CMPN, AND CMPP COMMANDS
32      SERVICE/IRDECOD:     --> U        [000172]      4698 PNL001; CONVERT PACKED, NUMERIC TO LONG COMMAND
32      SERVICE/IRDECOD:     --> U        [000173]      5279 VPNXX0; THE CONVERT PACKED TO NUMERIC COMMAND
32      SERVICE/IRDECOD:     --> U        [000174]      5495 VNPXX0;  THE CONVERT NUMERIC TO PACKED COMMAND
32      SERVICE/IRDECOD:     --> U        [000175]      4197 ASH001; THE ARITHMETIC SHIFT COMMAND, PACKED AND NUMERIC
32      SERVICE/IRDECOD:     --> U        [000176]      5733 LNP001; CONVERT LONG TO NUMERIC, PACKED COMMANDS
32      SERVICE/IRDECOD:     --> U        [000177]      6020 DVP001; THE MULTIPLY PACKED, AND DIVIDE PACKED COMMANDS
```

Figure 5-3 Sample Page of Microinstructions

```
.TOC    "MACROS FOR TEMPORARY RESULTS IN 2901-NO WRITE TO SCRATCH PAD(BPORT) NOR TO THE MBUS"

        TEMP_2901                 "ALUDST/LOADB2,CISSPW/SPNW,ENIB/ENIBN"
        T_OP[].RS_A[]B[]          "TEMP_2901,ALUSRC/AB,ALUFTN/01,APORT/02,BPORT/03"
        T_OP[],RS_A[]Q            "TEMP_2901,ALUSRC/AQ,ALUFTN/01,APORT/02"
        T_OP[],RS_0Q              "TEMP_2901,ALUSRC/,0Q,ALUFTN/01"
        T_OP[].RS_0B[]            "TEMP_2901,ALUSRC/,0B,ALUFTN/01,BPORT/02"
        T_OP[],RS_0A[]            "TEMP_2901,ALUSRC/.0A,ALUFTN/01,APORT/02"
        T_OP[],RS_D[]A[]          "TEMP_2901,ALUSRC/DA,ALUFTN/01,SALUI/02,APORT/03"
        T_OP[].RS_D[]Q            "TEMP_2901,ALUSRC/DQ,ALUFTN/01,SALUI/02"
        T_A[]                     "T_OP[R.OR.S].RS_0A[01]"
        T_A[].MINUS.1             "T_OP[S.MINUS,R].RS_0A[01]"
        T_A[].MINUS.B[]           "T_OP[R.MINUS.S].RS_A[01]B[02],ALUCB/YESCAR"
        T_A[].MINUS.B[].MINUS.1   "T_OP[R.MINUS.S].RS_A[01]B[02]"
        T_B[].MINUS.A[]           "T_OP[S.MINUS.R].RS_A[02]B[01],ALUCB/YESCAR"
```

Figure 5-4   Sample Page of Macrodefinitions

5.4 THE CIS MICROCODE INSTRUCTIONS

Each CIS instruction uses a group of words in the microstore. The number of words may be as few as in the L2Dn instruction or as many as in the DIVP instruction. This group of words (routine) may be completely self-contained or, when necessary, may call other routines or subroutines.

All instructions other than those for the L2Dn and L3Dn have a register and an in-line form. A large percentage of the microwords in the register form of the instruction are used by the in-line form.

Since all instructions except L2Dn and L3Dn (Appendix A) are suspendable, they have multiple start and resume or restart microword entry points. After suspension a "restore from interrupt" subroutine is executed to restore the instruction data so that the instruction can be completed.

Each microword in the KE44-A instruction set addresses a word in the KD11-Z microstore. To fully interpret the action of the KE44-A microword requires reading the word addressed in the KD11-Z.

At the end of each CIS instruction the KE44-A is returned to the idle loop.

CHAPTER 6
INSTALLATION AND CHECKOUT

6.1 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 6-1). 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 (toward the center of the module) for normal operation (Figure 6-2).

NOTE

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

ROWS

SLOTS — FRONT

	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			

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 6-1 Module Placement in Processor Backplane

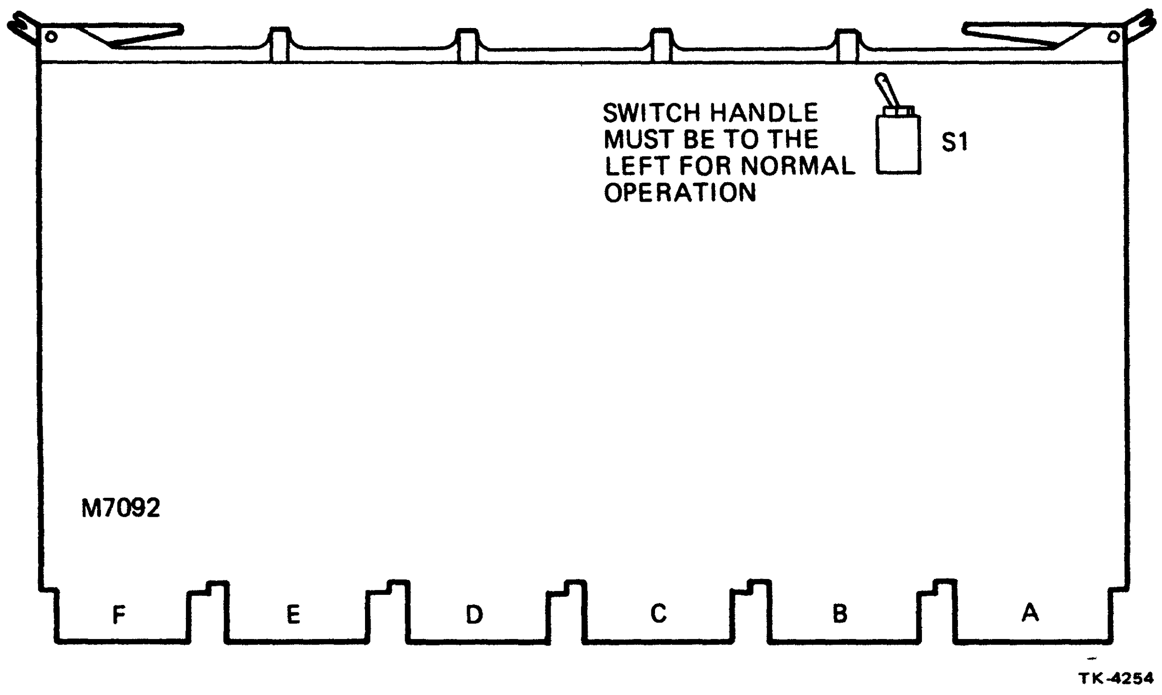


Figure 6-2 KE44-A Data Path/Logic Module, M7092

6.2 CHECKOUT

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

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

CHAPTER 7
MAINTENANCE

7.1 GENERAL

This chapter describes the use of the CZKEEA diagnostic program and the ASCII programmer's console in the maintenance of the KE44-A commercial instruction set option.

7.2 KE44-A DIAGNOSTICS

The CZKEEA is the only field diagnostic program available for the validation and diagnosis of the KE44-A. However, since the KD11-Z data path is used extensively in executing CIS instructions, CPU tests should be run prior to running CIS diagnostics if there is any doubt about the operational status of the CPU. However, successful running of the CPU tests does not rule out the possibility that a KD11-Z failure may cause only the CIS instructions to fail.

7.2.1 CZKEEA Program Abstract

The CIS instruction exerciser tests all CIS instructions in both register and in-line modes. Each instruction is tested:

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

7.2.2 Program Starting Procedure

The normal program starting address is 200. An optional starting address (204) provides for user selection of test instructions and control over the test environment. Another optional starting address (210) provides a quick-verify mode tailored to the type of processor under test. This mode has a run time of less than five minutes per pass and provides a fair level of microcode coverage (>80%).

7.2.2.1 Starting Address 200 – When the diagnostic CZKEEA is started at its normal starting address of 200, the execution (approximately 30 minutes on the PDP-11/44) of all tabled test cases for all instructions is followed by an "end-of-pass" indication. Testing then proceeds in a random mode until the operator terminates program execution.

CIS instruction interruptability will automatically be exercised if the system under test has either a line-time clock (KW11-L type) or a programmable real-time clock (KW11-P). The program uses the KW11-P at a frequency of 100 kHz if both clocks exist.

Processor mode (kernel, supervisor, user) is selected randomly prior to the execution of each test case in the CIS instructions. Memory management is enabled with the D-space enable/disable state selected randomly prior to each test case. Mode is switched to the test mode and memory management is turned on just prior to execution of the CIS instruction under test. During interrupt service, and immediately following the completion of the CIS instruction execution, the mode is switched back to kernel and memory management is shut off.

Tabled test cases are exhausted for any given instruction before proceeding to test the next CIS instruction. At the start of each new instruction in nonrandom mode, a message identifying the CIS instruction under test is displayed as a progress indicator. The following list gives the order in which instructions are tested in nonrandom mode, and the approximate number of tests executed for each instruction.

Instruction	Number of Tests
L2D	8
L3D	8
MOVC	354
LOCC	36
CMPC	362
MOVRC	354
MOVTC	354
SKPC	30
MATC	904
SCANC	126
SPANC	126
CVTPN	226
CVTNP	568
CVTLP	170
CVTLN	323
CVTPL	53
CVTNL	99
ADDP	1970
ADDN	3872
SUBP	1970
SUBN	3746
CMPP	502
CMPN	1089
ASHP	1972
ASHN	3872
MULP	1993
DIVP	1973

After being started at location 200, the program should respond as follows:

```
CZKEEA0 PDP-11 CIS instruction exerciser
Inst under test will be displayed.......
Pass time: 11/XX approx. XX min
L2D0 Inst Ct: XX XXXXX
          .
          .
          .
DIVP Inst Ct: XX XXXXX
End of pass (execution of tabled test cases complete)
Entering random test mode
No further end of pass messages will be issued
Random # generator seed constants will be printed
          Every 2000 CIS instruction tests
Random # generator seed XXXXXX XXXXXX XXXXXX
          .
          .
          .
(Until program execution is terminated by user)
```

A Control (^T) command entered at any time will cause the program to display the instruction under test and the current instruction count.

The instruction count displayed at the start of testing for each instruction is cumulative from the first L2D0 CIS instruction tested. The lower five digit count gets incremented once per executed CIS instruction test and counts from 0 to 65,535 (decimal). The upper two digit count gets incremented once per 65,535 tests. The instruction count is zeroed at the start of random mode testing. Control T must be used to display the instruction count in random mode.

7.2.2.2 Starting Address 204 – If the CZEEKA program is at address 204, the operator is required to respond to questions relating to the selection of instructions for test, test mode, and test environment.

After being started at location 204, the program should respond as follows:

```
CZKEEA0 PDP-11 CIS instruction exerciser
Test interruptability of CIS instructions (Y or N)?
Random exercise mode (Y or N)?
Enter instruction to test <All>
```

If the user answers yes (Y) to the interruptability question, the program will prompt for the selection of an interrupt source (e.g., the line-time clock (LTC); KW11-P at 100 kHz; KW11-P at 10 kHz; or KW11-P with external 1-MHz oscillator). If the LTC is selected, the program controls interrupt timing to assure that most CIS instructions are interrupted once. If the KW11-P with a 1-MHz external oscillator is selected, each CIS instruction will be interrupted and forced to suspend execution at all possible service exit points.

If either the KW11-P at 100 kHz or the KW11-P with external 1-MHz oscillator is selected, the program will ask whether or not to allow an interrupt during the CIS instruction DIVP (state disturbing instruction) normally executed within the KW11-P interrupt service routine.

If the user answers yes (Y) to the random exercise mode question, then the memory management test state, the processor test mode, test operands and string data for each CIS instruction test will be derived using a random number generator. A no (N) answer will cause execution of CIS instruction tests with all test operands and string data provided from program input and parameter tables. Following a (N) response, the program will prompt for processor test mode (kernel, supervisor, user) and memory management test state (off when D space is enabled, or on when D space is disabled).

The last question enables the user to select one or all CIS instructions for test. To select a single instruction for test, the mnemonic for the desired instruction is entered from the instruction list. The same question is repeated if the instruction is incorrectly entered. To select all CIS instructions for test (the default case) the operator simply responds with a carriage return.

If the random mode question is answered yes (Y) and the instruction(s) for test is/are answered by a <CR> indicating all, the actual instruction under test at any given point on the procedure is selected at random.

7.2.2.3 Starting Address 210 – If the diagnostic run is started at address 210, a quick verify (QV) pass provides a fair (more than 80 percent) level of microcode coverage in less than five minutes per pass.

This QV mode results in execution of a subset of the tabled test cases. The subset has been verified to provide at least the desired 80 percent level of coverage. Note that some CIS instructions may not be executed at all in QV mode, because it has been determined that, due to common routines within the microcode implementation, it is possible to get the desired 80 percent coverage without exercising all instructions.

The instruction counts listed above under the normal run mode (starting address 200) do not apply in QV mode.

CIS instruction interruptability is exercised provided that the system under test has either a line-time clock or a KW11-P programmable real-time clock.

Processor test mode (kernel, supervisor, user) and memory management test state are selected randomly as in the "starting address = 200" section above.

After being started at location 210, the program should respond as follows:

```
CZKEEA0 PDP-11 CIS instruction exerciser
Quick verify pass time: less than 5 minutes
L2D0 Inst CT: XX XXXXX
        .
        .
        .
DIVP Inst CT: XX XXXXX
End of quick verify pass
```

Random mode exercising is not invoked during a quick verify pass.

7.2.3 Error Information
If the computer halts without an error display, the following locations should be examined to determine information about the failing test.

TINST – CIS instruction under test
TR0 – TR6 – CIS instruction operands (lengths, addresses, etc.)

The information displayed upon detection of an error describes the complete environment of the failure. All instruction errors are displayed in one format. The format has slight variations to account for differences between character and decimal string instruction. Continuing the program from a trap will provide the user with a complete error printout.

7.2.4 Program Options
The following control characters are recognized by the exerciser during test execution:

CNTL T – Display instruction under test and test number.
CNTL C – Restart exerciser (recognized only if program was started at 204.)
CNTL D – Display all test case operands and results prior to each CIS instruction test.
CNTL E – Display all test case operands and results prior to each CIS instruction test. Query for continue.
CNTL N – Cancel prior CNTL D or CNTL E request.
CNTL O – Control over progress indication printout (i.e. INST and instruction CNT; random number generator seed; ON – OFF toggle).

7.2.5 Program Execution Times
For the PDP-11/44, first pass run time (tabled test cases only) is approximately 30 minutes.

After the first pass, the program enters random test mode and executes randomly generated test cases indefinitely.

In quick verify (QV) mode, pass time is less than five minutes.

7.3 ASC11 PROGRAMMER CONSOLE

The normal maintenance features provided by the programmer console for use in debugging and diagnosing the KD11-Z processor are directly extendable to the KE44-A CIS option. These features include:

- The console functions of examining and depositing data into the memory and general registers
- Single-instruction stepping
- Console maintenance features of single microinstruction stepping
- The displaying of MPC lines, UNIBUS data, CIS data and the contents of the machine dependent register.

The console displays MPC 0-10 L if the proper command is selected at the programmer console. Thus, single microstepping of the machine through the CIS microcode is possible.

A change in the KD11-Z processor (from its KD11-E predecessor) enables the AMUX lines onto the UNIBUS data lines.

NOTE

Refer to the PDP-11/44 Serial Console Specification for other details of console use.

NOTE

Appendix A has 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 DEFINITIONS

5.1 ADDN / ADDP / ADDNI / ADDPI - Add Decimal

Format:

```
             15            9 8         3 2   0
             ----------------------------------
      ADDN   |      076      |     05    | 0 |
             ----------------------------------


             ----------------------------------
      ADDP   |      076      |     07    | 0 |
             ----------------------------------


             ----------------------------------
     ADDNI   |      076      |     15    | 0 |
             ----------------------------------
             |          src1.dscr.ptr          |
             ----------------------------------
             |          src2.dscr.ptr          |
             ----------------------------------
             |          dst.dscr.ptr           |
             ----------------------------------

             ----------------------------------
     ADDPI   |      076      |     17    | 0 |
             ----------------------------------
             |          src1.dscr.ptr          |
             ----------------------------------
             |          src2.dscr.ptr          |
             ----------------------------------
             |          dst.dscr.ptr           |
             ----------------------------------
```

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:

Src1 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:

```
        15                              0
       ----------------------------------
   R0  |                                |
       ---          src1.dscr         ---
   R1  |                                |
       ----------------------------------
   R2  |                                |
       ---          src2.dscr         ---
   R3  |                                |
       ----------------------------------
   R4  |                                |
       ---          dst.dscr          ---
   R5  |                                |
       ----------------------------------
```

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

```
        15                              0
       ----------------------------------
   R0  |                0               |
       ----------------------------------
   R1  |                0               |
       ----------------------------------
   R2  |                0               |
       ----------------------------------
   R3  |                0               |
       ----------------------------------
   R4  |                                |
       ---          dst.dscr          ---
   R5  |                                |
       ----------------------------------
```

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. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Three Address Add - Register Form

```
MOV     SRC1.DSCR,RØ        ; 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,RØ         ; 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:

```
              15            9 8         3 2   0
              ----------------------------------
        ASHN  |     076       |    05     |  6  |
              ----------------------------------


              ----------------------------------
        ASHP  |     076       |    07     |  6  |
              ----------------------------------


              ----------------------------------
       ASHNI  |     076       |    15     |  6  |
              ----------------------------------
              |          src.dscr.ptr           |
              ----------------------------------
              |          dst.dscr.ptr           |
              ----------------------------------
              |           shift.dscr            |
              ----------------------------------


              ----------------------------------
       ASHPI  |     076       |    17     |  6  |
              ----------------------------------
              |          src.dscr.ptr           |
              ----------------------------------
              |          dst.dscr.ptr           |
              ----------------------------------
              |           shift.dscr            |
              ----------------------------------
```

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:

```
  15    12 11    8 7             0
 ----------------------------------
 |    0   |rnd.dgt|   shift.cnt    |
 ----------------------------------
```

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:

```
         15                              0
         ---------------------------------
    R0   |                               |
         ---          src.dscr         ---
    R1   |                               |
         ---------------------------------
    R2   |                               |
         ---          dst.dscr         ---
    R3   |                               |
         ---------------------------------
    R4   |          shift.dscr           |
         ---------------------------------
```

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

```
         15                              0
         ---------------------------------
    R0   |               0               |
         ---------------------------------
    R1   |               0               |
         ---------------------------------
    R2   |                               |
         ---          dst.dscr         ---
    R3   |                               |
         ---------------------------------
    R4   |               0               |
         ---------------------------------
```

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:

```
           15            9 8 7       3 2   0
          ----------------------------------
    CMPC  |      076      |     04    |  4 |
          ----------------------------------


          ----------------------------------
    CMPCI |      076      |     14    |  4 |
          ----------------------------------
          |          src1.dscr.ptr         |
          ----------------------------------
          |          src2.dscr.ptr         |
          ----------------------------------
          |        0        |     fill     |
          ----------------------------------
```

Operation:

Src1 is compared with src2 (src1-src2).

Condition Codes:

The condition codes are based on the arithmetic comparison of the most significant pair of unequal src1 and src2 characters (src1.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, src1.byte<7> and src2.byte<7> were different, and src2.byte<7> was the same as bit <7> of (src1.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 src1 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 R0-R1, the second source character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, and R4<15:8> must be zero:

```
          15               8 7              0
          -----------------------------------
     R0   |                                 |
          ---          src1.dscr          ---
     R1   |                                 |
          -----------------------------------
     R2   |                                 |
          ---          src2.dscr          ---
     R3   |                                 |
          -----------------------------------
     R4   |        0         |      fill    |
          -----------------------------------
```

The instruction terminates with sub-string descriptors in R0-R1 and R2-R3 which represent the portion of each source character string beginning with the most significant corresponding unequal characters. R0-R1 contain a descriptor for the unequal portion of the original src1 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.

```
          15               8 7              0
          -----------------------------------
     R0   |                                 |
          ---        sub.src1.dscr        ---
     R1   |                                 |
          -----------------------------------
     R2   |                                 |
          ---        sub.src2.dscr        ---
     R3   |                                 |
          -----------------------------------
     R4   |        0         |      fill    |
          -----------------------------------
```

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. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

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

    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:Ø>;      !    .
    R7 = R7+2;              !    .

    found = 1;
    while (src1.len nequ Ø) and (src2.len nequ Ø)
              and (found nequ Ø) 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 = Ø;
    while (src1.len nequ Ø) and (found nequ Ø) do
              if M[src1.adr] eqlu fill then
                  begin
                  src1.len = src1.len-1;
                  src1.adr = src1.adr+1
                  end
              else found = Ø;
    while (src2.len nequ Ø) and (found nequ Ø) 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     RØ,R2              ; length of shorter
        BHI     1$
        MOV     RØ,R2
    1$: MOV     R2,RØ
                                   ; 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 src1 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 src1. 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:

```
            15          9 8        3 2   0
           -------------------------------
    CMPN   |      076     |    05    | 2 |
           -------------------------------


           -------------------------------
    CMPP   |      076     |    07    | 2 |
           -------------------------------


           -------------------------------
   CMPNI   |      076     |    15    | 2 |
           -------------------------------
           |         src1.dscr.ptr       |
           -------------------------------
           |         src2.dscr.ptr       |
           -------------------------------


           -------------------------------
   CMPPI   |      076     |    17    | 2 |
           -------------------------------
           |         src1.dscr.ptr       |
           -------------------------------
           |         src2.dscr.ptr       |
           -------------------------------
```

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:

```
         15                              0
         ---------------------------------
    R0   |                               |
         ---          src1.dscr        ---
    R1   |                               |
         ---------------------------------
    R2   |                               |
         ---          src2.dscr        ---
    R3   |                               |
         ---------------------------------
```

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

```
         15                              0
         ---------------------------------
    R0   |               0               |
         ---------------------------------
    R1   |               0               |
         ---------------------------------
    R2   |               0               |
         ---------------------------------
    R3   |               0               |
         ---------------------------------
```

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:

```
            15           9 8        3 2  0
            ------------------------------
    CVTLN   |     076     |    05    | 7 |
            ------------------------------


            ------------------------------
    CVTLP   !     076     |    07    | 7 |
            ------------------------------


            ------------------------------
   CVTLNI   |     076     |    15    | 7 |
            ------------------------------
            |         dst.dscr.ptr       |
            ------------------------------
            |         src.long.ptr       |
            ------------------------------


            ------------------------------
   CVTLPI   |     076     |    17    | 7 |
            ------------------------------
            |         dst.dscr.ptr       |
            ------------------------------
            |         src.long.ptr       |
            ------------------------------
```

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 RØ-R1, and the source long integer is placed in R2-R3:

```
      15                              Ø
     -----------------------------------
RØ   |                                 |
     ---           dst.dscr          ---
R1   |                                 |
     -----------------------------------
R2   |                                 |
     ---           src.long          ---
R3   |                                 |
     -----------------------------------
```

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

```
      15                              Ø
     -----------------------------------
RØ   |                                 |
     ---           dst.dscr          ---
R1   |                                 |
     -----------------------------------
R2   |                Ø                |
     -----------------------------------
R3   |                Ø                |
     -----------------------------------
```

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. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Convert Long to Decimal - Register Form

```
MOV      DST.DSCR,R0       ; destination descriptor
MOV      DST.DSCR+2,R1
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:

```
               15           9 8        3 2   0
               ------------------------------------
       CVTNL   |      076      |    05    |  3  |
               ------------------------------------


               ------------------------------------
       CVTPL   |      076      |    07    |  3  |
               ------------------------------------


               ------------------------------------
       CVTNLI  |      076      |    15    |  3  |
               ------------------------------------
               |           src.dscr.ptr         |
               ------------------------------------
               |           dst.long.ptr         |
               ------------------------------------


               ------------------------------------
       CVTPLI  |      076      |    17    |  3  |
               ------------------------------------
               |           src.dscr.ptr         |
               ------------------------------------
               |           dst.long.ptr         |
               ------------------------------------
```

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 RØ-Rl:

```
         15                               Ø
         ----------------------------------
     RØ  |                                |
         ---          src.dscr          ---
     R1  |                                |
         ----------------------------------
```

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

```
         15                               Ø
         ----------------------------------
     RØ  |                Ø               |
         ----------------------------------
     R1  |                Ø               |
         ----------------------------------
     R2  |                                |
         ---          dst.long          ---
     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. RØ-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,R1
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:

```
                 15           9 8         3 2   0
                 ----------------------------------
        CVTNP   |     076      |     05    |  5  |
                 ----------------------------------


                 ----------------------------------
        CVTPN   |     076      |     05    |  4  |
                 ----------------------------------


                 ----------------------------------
        CVTNPI  |     076      |     15    |  5  |
                 ----------------------------------
                |          src.dscr.ptr          |
                 ----------------------------------
                |          dst.dscr.ptr          |
                 ----------------------------------


                 ----------------------------------
        CVTPNI  |     076      |     15    |  4  |
                 ----------------------------------
                |          src.dscr.ptr          |
                 ----------------------------------
                |          dst.dscr.ptr          |
                 ----------------------------------
```

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 R0-R1, and the destination descriptor is placed in R2-R3:

```
       15                              0
       -----------------------------------
R0     |                                 |
       ---           src.dscr          ---
R1     |                                 |
       -----------------------------------
R2     |                                 |
       ---           dst.dscr          ---
R3     |                                 |
       -----------------------------------
```

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

```
       15                              0
       -----------------------------------
R0     |                0                |
       -----------------------------------
R1     |                0                |
       -----------------------------------
R2     |                                 |
       ---           dst.dscr          ---
R3     |                                 |
       -----------------------------------
```

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. R0-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,R0        ; 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:

```
             15              9 8          3 2    0
             ----------------------------------
     DIVP    |       076       |      07    |  5  |
             ----------------------------------


             ----------------------------------
     DIVPI   |       076       |      17    |  5  |
             ----------------------------------
             |          src1.dscr.ptr           |
             ----------------------------------
             |          src2.dscr.ptr           |
             ----------------------------------
             |          dst.dscr.ptr            |
             ----------------------------------
```

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:

```
       15                               Ø
       ----------------------------------
    RØ |                                |
       ---         src1.dscr          ---
    R1 |                                |
       ----------------------------------
    R2 |                                |
       ---         src2.dscr          ---
    R3 |                                |
       ----------------------------------
    R4 |                                |
       ---          dst.dscr          ---
    R5 |                                |
       ----------------------------------
```

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

```
       15                               Ø
       ----------------------------------
    RØ |                Ø               |
       ----------------------------------
    R1 |                Ø               |
       ----------------------------------
    R2 |                Ø               |
       ----------------------------------
    R3 |                Ø               |
       ----------------------------------
    R4 |                                |
       ---          dst.dscr          ---
    R5 |                                |
       ----------------------------------
```

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,R1
        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:

```
             15            9 8 7      3 2   0
             ------------------------------------
      LOCC   |      076      |     04     |  0  |
             ------------------------------------


             ------------------------------------
     LOCCI   |      076      |     14     |  0  |
             ------------------------------------
             |          src.dscr.ptr            |
             ------------------------------------
             |        0        |     char       |
             ------------------------------------
```

Operation:

Search source character string for a character.

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 the search character. A character string descriptor is returned in R0-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 R0.

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:

```
          15            8 7             0
         ------------------------------------
     R0  |                                  |
         ---           src.dscr           ---
     R1  |                                  |
         ------------------------------------


         ------------------------------------
     R4  |          0       |      char     |
         ------------------------------------
```

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:

```
          15            8 7             0
         ------------------------------------
     R0  |                                  |
         ---         sub.src.dscr         ---
     R1  |                                  |
         ------------------------------------


         ------------------------------------
     R4  |          0       |      char     |
         ------------------------------------
```

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:

```
          15              8 7               0
         ------------------------------------
      R0 |                                  |
         ---         sub.src.dscr         ---
      R1 |                                  |
         ------------------------------------
```

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           ; RØ 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 RØ-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 RØ.

5.10 L2Dr - Load 2 Descriptors

Format:

```
            15          9 8          3 2   0
            ----------------------------------
     L2Dr   |     076     |     02     | r  |
            ----------------------------------
```

Operation:

Load word pairs into R0-R1 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 R0-R1; 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 RØ-R1 and the 'beta' descriptor is in R2-R3:

```
          15                               Ø
          ---------------------------------
     RØ   |                               |
          ---          alpha.dscr       ---
     R1   |                               |
          ---------------------------------
     R2   |                               |
          ---          beta.dscr        ---
     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;
    RØ = 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:

```
              15          9 8        3 2   0
             -----------------------------------
       L3Dr  |      076     |    06    |  r  |
             -----------------------------------
```

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 RØ-R1, the 'beta' descriptor is in R2-R3 and the 'gamma' descriptor is in R4-R5:

```
         15                               Ø
        -----------------------------------
    RØ  |                                 |
        ---          alpha.dscr         ---
    R1  |                                 |
        -----------------------------------
    R2  |                                 |
        ---          beta.dscr          ---
    R3  |                                 |
        -----------------------------------
    R4  |                                 |
        ---          gamma.dscr         ---
    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;
    RØ = 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:

```
            15           9          3 2   0
           ----------------------------------
   MATC    |      076      |      04    |  5  |
           ----------------------------------


           ----------------------------------
   MATCI   |      076      |      14    |  5  |
           ----------------------------------
           |           src.dscr.ptr          |
           ----------------------------------
           |           obj.dscr.ptr          |
           ----------------------------------
```

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-R1, and the object character string descriptor is placed in R2-R3:

```
        15                              Ø
        ----------------------------------
   RØ   |                                |
        ---          src.dscr          ---
   R1   |                                |
        ----------------------------------
   R2   |                                |
        ---          obj.dscr          ---
   R3   |                                |
        ----------------------------------
```

The instruction terminates with a character sub-string descriptor returned in RØ-R1 which represents the portion of the original source character string beginning with the most significant character to completely match the object character string.

```
        15                              Ø
        ----------------------------------
   RØ   |                                |
        ---        sub.src.dscr        ---
   R1   |                                |
        ----------------------------------
   R2   |                                |
        ---          obj.dscr          ---
   R3   |                                |
        ----------------------------------
```

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-R1 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.

```
         15                             Ø
         ----------------------------------
    RØ   |                                |
         ---          sub.src.dscr      ---
    Rl   |                                |
         ----------------------------------
```

Formal Description:

```
    src.len = RØ;          ! MATC only
    src.adr = Rl;          !    .
    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 = Ø;
    while (src.len gequ obj.len) and (obj.len nequ Ø)
      and (found eqlu Ø) do
             begin
             same = 1;
             while (obj.len nequ Ø) 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 = Ø;
             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
             RØ = src.len+1;
             Rl = src.adr-1
             end
```

```
else
        begin
        RØ = Ø;
        Rl = src.adr+src.len
        end;

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

N = RØ<15>;
Z = RØ eqlu Ø;
V = Ø;
C = Ø;
```

Examples:

1. Find a Keyword - Register Form

```
MOV     SRC.DSCR,RØ       ; lst source descriptor
MOV     SRC.DSCR+2,Rl
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 RØ-Rl.

3. If the length of the object character string is greater than that of the source character string then no match is found; RØ-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:

```
           15           9 8 7      3 2   0
          ---------------------------------
   MOVC   |     076      |     03    | 0 |
          ---------------------------------


          ---------------------------------
  MOVCI   |     076      |     13    | 0 |
          ---------------------------------
          |          src.dscr.ptr          |
          ---------------------------------
          |          dst.dscr.ptr          |
          ---------------------------------
          |       0        |     fill      |
          ---------------------------------
```

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:

```
         15              8 7            0
         ---------------------------------
    R0   |                               |
         ---          src.dscr         ---
    R1   |                               |
         ---------------------------------
    R2   |                               |
         ---          dst.dscr         ---
    R3   |                               |
         ---------------------------------
    R4   |      0        |      fill     |
         ---------------------------------
```

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

```
         15              8 7            0
         ---------------------------------
    R0   |     max(0,src.len-dst.len)    |
         ---------------------------------
    R1   |               0               |
         ---------------------------------
    R2   |               0               |
         ---------------------------------
    R3   |               0               |
         ---------------------------------
    R4   |      0        |      fill     |
         ---------------------------------
```

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. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

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

    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: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.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[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[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 = Ø;                !    .
R2 = Ø;                !    .
R3 = Ø;                !    .
R4 = Ø<15:8>@fill;     !    .
```

Examples:

1. Moving Data - Register Form

```
        MOV     SRC.DSCR,RØ      ; 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     R0              ; 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   0               ; 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, R0 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:

```
          15           9 8 7      3 2   0
          ---------------------------------
MOVRC     |     076     |    03    | 1 |
          ---------------------------------


          ---------------------------------
MOVRCI    |     076     |    13    | 1 |
          ---------------------------------
          |         src.dscr.ptr          |
          ---------------------------------
          |         dst.dscr.ptr          |
          ---------------------------------
          |      0       |     fill       |
          ---------------------------------
```

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 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:

```
          15                8 7                0
          ----------------------------------------
     R0   |                                      |
          ---             src.dscr             ---
     R1   |                                      |
          ----------------------------------------
     R2   |                                      |
          ---             dst.dscr             ---
     R3   |                                      |
          ----------------------------------------
     R4   |         0          |        fill     |
          ----------------------------------------
```

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

```
          15                8 7                0
          ----------------------------------------
     R0   |       max(0,src.len-dst.len)         |
          ----------------------------------------
     R1   |                  0                   |
          ----------------------------------------
     R2   |                  0                   |
          ----------------------------------------
     R3   |                  0                   |
          ----------------------------------------
     R4   |         0          |        fill     |
          ----------------------------------------
```

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 = R1;          !    .
    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
```

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.

```
                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 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;

    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
```

5.18 MOVTC / MOVTCI - Move Translated Character

Format:

```
              15           9 8 7        3 2   0
              ----------------------------------
      MOVTC   |     076      |      03    | 2  |
              ----------------------------------


              ----------------------------------
     MOCTCI   |     076      |      13    | 2  |
              ----------------------------------
              |          src.dscr.ptr          |
              ----------------------------------
              |          dst.dscr.ptr          |
              ----------------------------------
              |      0         |     fill      |
              ----------------------------------
              |           table.adr            |
              ----------------------------------
```

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<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 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 RØ-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:Ø>, R4<15:8> must be zero, and the translation table address is placed in R5:

```
        15            8 7           Ø
       -----------------------------------
  RØ   |                                 |
       ---          src.dscr           ---
  R1   |                                 |
       -----------------------------------
  R2   |                                 |
       ---          dst.dscr           ---
  R3   |                                 |
       -----------------------------------
  R4   |       Ø       |      fill       |
       -----------------------------------
  R5   |            table.adr            |
       -----------------------------------
```

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

```
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[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 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[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;

R0 = src.len;         ! MOVTC only
R1 = 0;               !    .
R2 = 0;               !    .
R3 = 0;               !    .
R4 = 0<15:8>@fill;    !    .
R5 = table.adr;       !    .
```

```
          15            8 7            0
         ---------------------------------
     R0  |     max(0,src.len-dst.len)    |
         ---------------------------------
     R1  |               0               |
         ---------------------------------
     R2  |               0               |
         ---------------------------------
     R3  |               0               |
         ---------------------------------
     R4  |       0       |      fill     |
         ---------------------------------
     R5  |           table.adr           |
         ---------------------------------
```

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;
```

Examples:

1. Character Code Conversion - Register Form

```
        MOV     SRC.DSCR,RØ         ; EBCDIC source
        MOV     SRC.DSCR+2,R1
        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:

```
               15            9 8          3 2   0
               ------------------------------------
        MULP   |       076       |     07    | 4 |
               ------------------------------------


               ------------------------------------
        MULPI  |       076       |     17    | 4 |
               ------------------------------------
               |          src1.dscr.ptr            |
               ------------------------------------
               |          src2.dscr.ptr            |
               ------------------------------------
               |          dst.dscr.ptr             |
               ------------------------------------
```

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:

Src1 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 R0-R1, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:

```
         15                              0
         ---------------------------------
     R0  |                               |
         ---          src1.dscr        ---
     R1  |                               |
         ---------------------------------
     R2  |                               |
         ---          src2.dscr        ---
     R3  |                               |
         ---------------------------------
     R4  |                               |
         ---          dst.dscr         ---
     R5  |                               |
         ---------------------------------
```

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

```
         15                              0
         ---------------------------------
     R0  |               0               |
         ---------------------------------
     R1  |               0               |
         ---------------------------------
     R2  |               0               |
         ---------------------------------
     R3  |               0               |
         ---------------------------------
     R4  |                               |
         ---          dst.dscr         ---
     R5  |                               |
         ---------------------------------
```

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:

```
          15            9 8 7      3 2  0
          -----------------------------------
 SCANC    |      076      |     04    | 2 |
          -----------------------------------


          -----------------------------------
 SCANCI   |      076      |     14    | 2 |
          -----------------------------------
          |          src.dscr.ptr           |
          -----------------------------------
          |          set.dscr.ptr           |
          -----------------------------------
```

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-R1, and the character set descriptor is placed in R4-R5:

```
       15                              Ø
       ---------------------------------
  RØ   |                               |
       ---          src.dscr         ---
  R1   |                               |
       ---------------------------------


       ---------------------------------
  R4   |                               |
       ---          set.dscr         ---
  R5   |                               |
       ---------------------------------
```

When the instruction is completed, RØ-R1 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:

```
       15                              Ø
       ---------------------------------
  RØ   |                               |
       ---        sub.src.dscr       ---
  R1   |                               |
       ---------------------------------


       ---------------------------------
  R4   |                               |
       ---          set.dscr         ---
  R5   |                               |
       ---------------------------------
```

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-R1 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:

```
             15                                   0
            ------------------------------------
        R0  |                                  |
            ---          sub.src.dscr        ---
        R1  |                                  |
            ------------------------------------
```

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-R1.

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:

```
              15          9 8 7      3 2   Ø
             ----------------------------------
       SKPC  |       Ø76      |     Ø4    | 1 |
             ----------------------------------


             ----------------------------------
      SKPCI  |       Ø76      |     14    | 1 |
             ----------------------------------
             |         src.dscr.ptr           |
             ----------------------------------
             |        Ø        |     char     |
             ----------------------------------
```

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-R1, the search character is placed in R4<7:Ø>, and R4<15:8> must be zero:

```
         15             8 7             Ø
        -----------------------------------
   RØ   |                                 |
        ---          src.dscr           ---
   R1   |                                 |
        -----------------------------------


        -----------------------------------
   R4   |        Ø        |      char     |
        -----------------------------------
```

When the instruction is completed, RØ-R1 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:

```
         15             8 7             Ø
        -----------------------------------
   RØ   |                                 |
        ---        sub.src.dscr         ---
   R1   |                                 |
        -----------------------------------


        -----------------------------------
   R4   |        Ø        |      char     |
        -----------------------------------
```

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-R1 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:

```
        15                              Ø
        ---------------------------------
    RØ  |                               |
        ---         sub.src.dscr      ---
    R1  |                               |
        ---------------------------------
```

Formal Description:

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

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

    found = 1;
    while (src.len nequ Ø) 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 = Ø;

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

    N = RØ<15>;
    Z = RØ eqlu Ø;
    V = Ø;
    C = Ø;
```

Examples:

1. Skip Leading Spaces - Register Form

```
        MOV     STR.DSCR,RØ      ; 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 RØ-R1.

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

5.22 SPANC / SPANCI - Span Character

Format:

```
             15            9 8 7       3 2   0
             ------------------------------------
       SPANC |       076      |     04     | 3  |
             ------------------------------------


             ------------------------------------
      SPANCI |       076      |     14     | 3  |
             ------------------------------------
             |          src.dscr.ptr            |
             ------------------------------------
             |          set.dscr.ptr            |
             ------------------------------------
```

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 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 character which is not a member of the character set. A character string descriptor is returned in RØ-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 RØ.

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 R0-R1, and the character set descriptor is placed in R4-R5:

```
          15                              0
          ----------------------------------
     R0   |                                |
          ---          src.dscr          ---
     R1   |                                |
          ----------------------------------


          ----------------------------------
     R4   |                                |
          ---          set.dscr          ---
     R5   |                                |
          ----------------------------------
```

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 character which is not a member of the character set:

```
          15                              0
          ----------------------------------
     R0   |                                |
          ---        sub.src.dscr        ---
     R1   |                                |
          ----------------------------------


          ----------------------------------
     R4   |                                |
          ---          set.dscr          ---
     R5   |                                |
          ----------------------------------
```

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, R0-R1 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:

```
            15                          Ø
            ----------------------------------
        RØ  |                                |
            ---        sub.src.dscr        ---
        R1  |                                |
            ----------------------------------
```

Formal Description:

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

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

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

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

    N = RØ<15>;
    Z = RØ eqlu Ø;
    V = Ø;
    C = Ø;
```

Examples:

1. Pass Tabs and Blanks - Register Form

```
        MOV     STR.DSCR,RØ         ; 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   Ø                   ; ASCII ØØØ
        .BYTE   Ø                   ; ASCII ØØ1
        .BYTE   Ø                   ; ASCII ØØ2
                                        .
                                        .
                                        .
        .BYTE   2                   ; ASCII Ø11 = TAB
        .BYTE   Ø                   ; ASCII Ø12
        .BYTE   Ø                   ; ASCII Ø13
                                        .
                                        .
                                        .
        .BYTE   2                   ; ASCII Ø4Ø = SPACE
        .BYTE   Ø                   ; ASCII Ø41
        .BYTE   Ø                   ; ASCII Ø42
                                        .
                                        .
                                        .
        .BYTE   Ø                   ; 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-R1.

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:

```
             15          9 8       3 2  0
             ----------------------------
    SUBN     |    076     |   05   | 1 |
             ----------------------------


             ----------------------------
    SUBP     |    076     |   07   | 1 |
             ----------------------------


             ----------------------------
   SUBNI     |    076     |   15   | 1 |
             ----------------------------
             |       src1.dscr.ptr     |
             ----------------------------
             |       src2.dscr.ptr     |
             ----------------------------
             |       dst.dscr.ptr      |
             ----------------------------


             ----------------------------
   SUBPI     |    076     |   17   | 1 |
             ----------------------------
             |       src1.dscr.ptr     |
             ----------------------------
             |       src2.dscr.ptr     |
             ----------------------------
             |       dst.dscr.ptr      |
             ----------------------------
```

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 RØ-Rl, the second source descriptor is placed in R2-R3, and the destination descriptor is placed in R4-R5:

```
       15                            Ø
      ----------------------------------
  RØ  |                                |
      ---          srcl.dscr         ---
  Rl  |                                |
      ----------------------------------
  R2  |                                |
      ---          src2.dscr         ---
  R3  |                                |
      ----------------------------------
  R4  |                                |
      ---          dst.dscr          ---
  R5  |                                |
      ----------------------------------
```

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

```
       15                            Ø
      ----------------------------------
  RØ  |               Ø                |
      ----------------------------------
  Rl  |               Ø                |
      ----------------------------------
  R2  |               Ø                |
      ----------------------------------
  R3  |               Ø                |
      ----------------------------------
  R4  |                                |
      ---          dst.dscr          ---
  R5  |                                |
      ----------------------------------
```

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. RØ-R6 are unchanged when the instruction is completed.

Formal Description:

TBS;

Examples:

1. Three address subtract - Register Form

```
        MOV     SRC1.DSCR,RØ       ; 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,RØ        ; 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 B
CIS MPC FUNCTIONS

740	R12_R12
741	R0L_R0L
742	R1L_R1L
743	R2L_R2L
744	R3L_R3L
745	R4L_R4L
746	R5L_R5L
747	R6L_R6L, ENAB STOV
750	R7L_R7L
751	BA,R6_R6-2, ENAB STOV
752	BA, R6_R6+2, ENAB STOV
753	R14_R14
754	R10_R10
755	R6_R6, ENAB STOV
756	R7_R7
757	R12L_R12L
760	R13L_R13L
761	R14L_R14L
762	R10L_R10L
763	PSW_PSW
764	BA_R6, DATI (D), B_UDATA
765	BA_R13, DATI (D), EXTERNAL_UDATA
766	BA_R14, DATI (D), EXTERNAL_UDATA
767	BA_R10, DATI (D), EXTERNAL_UDATA
770	BA_R10, DATI (D), B_UDATA
771	BA_R10, DATI (I), B_UDATA
772	BA_R6
773	BA_R10
774	DATO(D), UDATA_B
775	DATOB(D), UDATA_EXTERNAL
776	BA_R10L

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⟩

KE44-A CISP
TECHNICAL MANUAL
EK-KE44A-TM-001

Reader's Comments

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-TM

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