DEC STANDARD 168

PDP11

Extended

Instructions

TITLE: PDP11 Extended Instructions

DATE	ECO #	AUTHOR	APPROVED	REV	SEC	PAGES
18-Jan-79	--	Lloyd Dickman	C. Noelcke	A	--	All

Size	Code	Number	Rev.
A	DS	EL00168-00	A

DEC STANDARD 168

Revision A

PDP11 EXTENDED INSTRUCTIONS

18 January 1979

ABSTRACT

This standard provides architectural definition and control for PDP11 instructions whose opcodes are in the reserved and extended opcode spaces.

Revision History

Pass #	Description	Author	Revised Date
Pass 1	Original Standard	L. Dickman	30 August 1977
Pass 2	March 1978 Task Force	L. Dickman	23 August 1978

CHAPTER 1 INTRODUCTION

1.1 NATURE OF THIS STANDARD 1-2
1.1.1 Purpose 1-2
1.1.2 Scope 1-2
1.1.3 Motives 1-3
1.1.4 Applicability and Waivers 1-3
1.1.5 Goals 1-3
1.1.6 Non-Planned Effects of Goals 1-4
1.1.7 Non-Goals 1-4
1.1.8 History of Previous Standarization Efforts 1-4
1.1.9 Related Current Standards 1-4
1.1.10 Future Standards Activities 1-4
1.2 CHANGES TO THIS STANDARD (ECO'S) 1-4
1.3 FORMAL ISPS DEFINITIONS 1-5
1.4 POLICIES 1-5
1.4.1 PDP11 Family Compatibility 1-5
1.4.2 PDP11 / VAX11 Compatibility 1-5
1.4.3 Processor-Model Identification 1-5
1.4.4 Instruction-Group Atomicity 1-5

CHAPTER 2 FRAMEWORK FOR EXTENDING THE PDP11 INSTRUCTION SET

2.1 OPCODE UTILIZATION AND AVAILABILITY 2-2
2.2 OPCODE GROUPINGS 2-2
2.3 INSTRUCTION-STREAM CONTENTS 2-2
2.4 FORMAT OF OPCODE 2-3
2.5 EXTENDED-INSTRUCTION GROUPS 2-3
2.6 EXTENDED INSTRUCTION CATEGORIES 2-4
2.7 OPERANDS FOR EXTENDED INSTRUCTIONS 2-5
2.7.1 Implicit Operands 2-5
2.7.2 Explicit Opcode-Specific Operands 2-5
2.7.3 Explicit General Operands 2-6
2.7.3.1 Number and Types of General Operand Formats . . . 2-6
2.7.3.2 Format of Single-Operand Specifiers - 2-6
2.7.3.3 Format of Double-Operand Specifiers - 2-6
2.7.3.4 Additional Instruction Stream Operand Words . . . 2-7
2.8 TRAPS 2-7
2.8.1 Reserved-Instruction Traps -- Vector 10(8) 2-7
2.8.2 Trace Traps -- Vector 14(8) 2-8
2.8.3 Fatal Traps -- Vector 4(8) 2-8
2.8.4 Floating Point Traps -- Vector 244(8) 2-9
2.8.5 Other Traditional Traps 2-9
2.8.6 Traps Unique to Extended Instructions 2-9
2 SUSPENDABLE INSTRUCTIONS 2-9
2.9.1 Suspendability Classifications 2-10
2.9.2 Non-Suspendable Instructions 2-10
2.9.3 Potentially Suspendable Instructions 2-10
2.9.4 Instruction Suspension 2-11
2.10 STACK UTILIZATION 2-12
2.11 UNPREDICTABLE CONDITIONS 2-13

CHAPTER 2 FRAMEWORK FOR EXTENDING THE PDP11 INSTRUCTION SET (Con't)

2.12 RESERVATION OF UNUSED FIELDS AND ENCODINGS 2-13
2.13 MULTIPROGRAMMING INTEGRITY 2-14

CHAPTER 3 EXTENDED-INSTRUCTION DATA TYPES

3.1 CHARACTER DATA TYPES 3-2
3.1.1 Character 3-2
3.1.2 Character String 3-2
3.1.3 Character Set 3-3
3.2 DECIMAL STRING DATA TYPES 3-4
3.2.1 Common Properties 3-4
3.2.2 Decimal String Descriptors 3-6
3.2.3 Packed Strings 3-7
3.2.4 Zoned Strings 3-9
3.2.5 Overpunch Strings 3-10
3.2.6 Separate Strings 3-12
3.3 LONG INTEGER 3-15

CHAPTER 4 DESCRIPTION AND INTENT OF EXTENDED INSTRUCTIONS

4.1 PROCESSOR IDENTIFICATION INSTRUCTION 4-2
4.2 COMMERCIAL INSTRUCTION SET 4-2
4.2.1 Character String Instructions 4-3
4.2.1.1 Instructions - 4-3
4.2.1.1.1 Character String Move Instructions - 4-3
4.2.1.1.2 Character String Search Instructions 4-4
4.2.1.2 Condition Codes - 4-4
4.2.1.3 Operand Delivery - 4-5
4.2.1.4 Data Overlap - 4-6
4.2.1.5 Unpredictable Conditions - 4-6
4.2.1.6 Implementation Notes - 4-7
4.2.2 Decimal String Instructions 4-8
4.2.2.1 Instructions - 4-8
4.2.2.2 Condition Codes - 4-9
4.2.2.3 Operand Delivery - 4-10
4.2.2.4 Data Overlap - 4-10
4.2.2.5 Unpredictable Conditions - 4-11
4.2.2.6 Implementation Notes - 4-12
4.2.3 Commercial Load Descriptor Instructions 4-13
4.2.3.1 Implementation Notes - 4-14
4.3 PROCESSOR SPECIFIC INSTRUCTIONS 4-15

CHAPTER 5 EXTENDED-INSTRUCTION DEFINITIONS

5.1 ADDN / ADDP / ADDNI / ADDPI - ADD DECIMAL 5-2
5.2 ASHN / ASHP / ASHNI / ASHPI - ARITHMETIC SHIFT DE 5-6
5.3 CMPC / CMPCI - COMPARE CHARACTER 5-10
5.4 CMPN / CMPP / CMPNI / CMPPI - COMPARE DECIMAL 5-15
5.5 CVTLN / CVTLP / CVTLNI / CVTLPI - CONVERT LONG T 5-18
5.6 CVTNL / CVTPL / CVTNLI / CVTPLI - DECIMAL TO LON 5-21
5.7 CVTNP / CVTPN / CVTNPI / CVTPNI - CONVERT DECIMA 5-24
5.8 DIVP / DIVPI - DIVIDE DECIMAL 5-27
5.9 LOCC / LOCCI - LOCATE CHARACTER 5-30
5.10 L2DR - LOAD 2 DESCRIPTORS 5-34
5.11 L3DR - LOAD 3 DESCRIPTORS 5-36
5.12 MATC / MATCI - MATCH CHARACTER 5-39
5.13 MED6X - PDP11/60 MAINTENANCE, EXAMINE, DEPOSIT 5-44
5.14 MED74C - PDP11/74 CIS MAINTENANCE INSTRUCTION 5-47
5.15 MFPT - MOVE FROM PROCESSOR TYPE 5-49
5.16 MOVC / MOVCI - MOVE CHARACTER 5-51
5.17 MOVRC / MOVRCI - MOVE REVERSE JUSTIFIED CHARACT 5-56
5.18 MOVTC / MOVTCI - MOVE TRANSLATED CHARACTER 5-61
5.19 MULP / MULPI - MULTIPLY DECIMAL 5-66
5.20 SCANC / SCANCI - SCAN CHARACTER 5-69
5.21 SKPC / SKPCI - SKIP CHARACTER 5-74
5.22 SPANC / SPANCI - SPAN CHARACTER 5-78
5.23 SUBN / SUBP / SUBNI / SUBPI - SUBTRACT DECIMAL 5-83

CHAPTER 6 REINTERPRETATION OF TRADITIONAL PDP11 INSTRUCTIONS

6.1 MULTIPROCESSING MEMORY LOCK 6-2

CHAPTER 7 WAIVERS

7.1 PDP11/60 LACKING MFPT 7-2
7.2 LSI-11 COMMERCIAL INSTRUCTION SET 7-3

APPENDIX A EXTENDED-INSTRUCTION OPCODE ASSIGNMENTS

APPENDIX B PDP11 OPCODE SPACE

APPENDIX C FORMAL DESCRIPTION OF MACHINE STATE

CHAPTER 1

INTRODUCTION

1.1 NATURE OF THIS STANDARD

1.1.1 Purpose

The purpose of this Standard is to provide architectural definition and control for PDP11 instructions whose opcodes lie in the reserved and extended opcode spaces.[1]

1.1.2 Scope

The scope of this Standard covers all programmable aspects of PDP11 instructions in the reserved and extended opcode spaces. "Programmable aspects" include all aspects of the instructions which are controllable by, are visible to, or affect the behavior of PDP11 programs. The reserved and extended opcode spaces are defined in Section 2.1 and are enumerated in Appendix B.

Except as specified in Chapter 6, the scope of this Standard does not extend to the instructions historically established in the implementations of PDP11 processors prior to March 1976, because the definition of those instructions is fixed. Specifically this exclusion refers to the instructions implemented in the following models:

KA11-YA
KB11-A
KB11-B
KB11-C
KB11-D
KD11-A
KD11-B
KD11-D
KD11-E.

The exclusion of these historically established instructions from the scope of this Standard does not imply that freedom or latitude exists relative to their architectural definitions.

---

[1] The work leading to this standard is described in "PDP11 Instruction Extensions" by Lloyd Dickman, 1 March 1976, 8 pp.

1.1.3 Motives

This Standard is intended to provide designers with definitions that will ensure architectural consistency of new machine instructions across processors of the PDP11 ily. This will consequently promote the general transportability o software across members of the PDP11 system family, will reduce associated support problems in both the hardware and software areas, and will control the variability that might otherwise impede migration of software structures to the VAX11 family.

1.1.4 Applicability and Waivers

This standard applies to all PDP11 processors announced during or after March 1976 and to any major revision of a PDP11 processor.

Exceptions to this Standard will be documented in Chapter 7. The documentation must specify in detail both the extent of the exception and the reasons for the exception. The intended exception will then be reviewed by the PDP11 Architecture Manager, who will submit a written recommendation to the Engineering Committee that it either approve, reject or amend the proposed waiver. The Engineering Committee's decision shall be incorporated by the PDP11 Architecture Group Manager into this Standard.

1.1.5 Goals

The goals of this standard are:

1. to specify the framework within which new instructions can be added to the PDP11 architecture,

2. to serve as a centrally controlled repository for the specifications of all PDP11 extended instructions, and

3. to serve as a centrally controlled repository for all necessary re-interpretations of traditional PDP11 instructions.

1.1.6 Non-Planned Effects of Goals

The effects, both planned and non-planned, will be documented in the treatment of each extended instruction in the text of the standard.

1.1.7 Non-Goals

Non-Goals of this standard are:

1. This document does not attempt to define traditional PDP11 instructions, except as noted above in section 1.1.5, item 3.

2. This document does not attempt to plan or define specific future extensions to the PDP11 instruction set. Its intent is to define the framework within which such extensions can be made and to record the specifications of extended instructions that are actually implemented on PDP11 processors.

1.1.8 History of Previous Standarization Efforts

None.

1.1.9 Related Current Standards

None.

1.1.10 Future Standards Activities

None.

1.2 CHANGES TO THIS STANDARD (ECO'S)

The normal method for effecting changes to this Standard is to submit the proposed change in the form of an ECO to the PDP11 Architecture Manager for review and approval. The manager will send the proposed ECO, together with a recommendation to the Engineering Committee for final decision. The PDP11 Architecture Group Manager will incorporate approved ECO's into this Standard.

1.3 FORMAL ISPS DEFINITIONS

Formal descriptions in the ISPS language, when provided, are an essential part of an instruction's specification. They are included to specify, as accurately as possible, the architected results obtained from the execution of the instructions but do not necessarily imply implementation methods or algorithms. Where provided, the formal ISPS descriptions are the authorative source of information about the instructions; the English and pictorial descriptions serve a secondary role. The machine state for each of these descriptions appears in Appendix C. All ISPS statements use a semi-colon to signify synchronization. Thus, the semi-colon is an implied 'next' operator.

1.4 POLICIES

1.4.1 PDP11 Family Compatibility

In general, PDP11 extended instructions shall be so defined as to be implementable on any processor of the PDP11 family. Optimization of an instruction for a particular processor shall not preclude the possibility of its implementation on other processors of the PDP11 family. Exceptions to this policy are relegated to the processor-specific instruction groups (-- see Section 2.6).

1.4.2 PDP11/VAX11 Compatibility

Data types associated with extended PDP11 instructions shall be consistent with corresponding VAX11 data types. This will facilitate migration of data files from PDP11 systems to VAX11 systems.

1.4.3 Processor-Model Identification

Any major revision to an existent PDP11 processor and all new PDP11 processors will include implementation of the MFPT instruction (-- see Chapter 5).

1.4.4 Instruction-Group Atomicity

Implementors will provide either all or none of the instructions of a closed group (-- see Section 2.2).

CHAPTER 2

Framework for Extending the PDP11 Instruction Set

2.1 OPCODE UTILIZATION AND AVAILABILITY

Opcodes in the following ranges are reserved and are not available for usage:

```
000010(8) - 000077(8)
007000(8) - 007777(8)
107000(8) - 107777(8)
170006(8)
170010(8)
170013(8) - 170077(8)
```

In general, extended PDP11 instructions will utilize opcodes in the range 076000(8) - 076777(8).

2.2 OPCODE GROUPINGS

The extended opcode space is divided into 64 groups of 8 instructions each. Groups are treated as integral entities. A group is declared "closed" when all 8 instructions in it have been defined or when no further instructions are admissible into it. Otherwise a group is considered "open" and future instructions may be added into it. The opcode groups are specified in Section 2.5. See also Section 1.4.4.

2.3 INSTRUCTION-STREAM CONTENTS

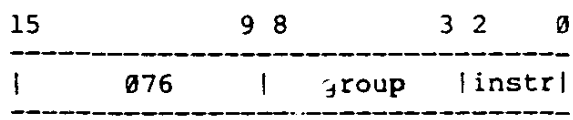
PDP11 extended instructions can be defined (a) to operate on implicitly specified operands and/or (b) to require explicit operand specifiers in the instruction stream. Explicit operand specifiers may use either (i) a general operand-specifier format or (ii) an opcode-specific operand-specifier format.

If an extended instruction uses only implicit operands, only the opcode will appear in the instruction-stream (-- see Section 2.7).

If an extended instruction uses explicit operands, the opcode word is followed in the instruction stream by as many operand specifiers and operands as the specification of the instruction requires. As in traditional PDP11 instructions, explicit general operand specifiers using modes 6 or 7 or using R7 in modes 2 or 3 will also require additional words in the instruction stream (-- see Section 2.7).

2.4 FORMAT OF OPCODE

The extended instruction opcode word is structured as follows:



Bits <8:3> contain the group code. Bits <2:0> specify the instruction within the group.

2.5 EXTENDED-INSTRUCTION GROUPS

The extended-instruction groups are defined in the following table, where X represents the set of eight instructions in the group.

CODE	GROUP	STATUS
07600X		open
07601X		open
07602X	Commercial Load 2 Descriptors	closed
07603X	Character String Move	closed
07604X	Character String Search	closed
07605X	Numeric String	closed
07606X	Commercial Load 3 Descriptors	closed
07607X	Packed String	closed
07610X		open
07611X		open
07612X		open
07613X	Character String Move (in-line)	closed
07614X	Character String Search (in-line)	closed
07615X	Numeric String (in-line)	closed
07616X		open
07617X	Packed String (in-line)	closed
07620X		open
07621X		open
07622X		open
07623X		open
07624X		open
07625X		open
07626X		open
07627X		open
07630X		open
07631X		open
07632X		open
07633X		open
07634X		open

CODE	GROUP	STATUS
07635X		open
07636X		open
07637X		open
07640X		open
07641X		open
07642X		open
07643X		open
07644X		open
07645X		open
07646X		open
07647X		open
07650X		open
07651X		open
07652X		open
07653X		open
07654X		open
07655X		open
07656X		open
07657X		open
07660X	Processor-Specific #0	open
07661X	Processor-Specific #1	open
07662X	Processor-Specific #2	open
07663X	Processor-Specific #3	open
07664X	Processor-Specific #4	open
07665X	Processor-Specific #5	open
07666X	Processor-Specific #6	open
07667X	Processor-Specific #7	open
07670X	CSS/Customer #0	open
07671X	CSS/Customer #1	open
07672X	CSS/Customer #2	open
07673X	CSS/Customer #3	open
07674X	CSS/Customer #4	open
07675X	CSS/Customer #5	open
07676X	CSS/Customer #6	open
07677X	CSS/Customer #7	open

2.6 EXTENDED INSTRUCTION CATEGORIES

The extended instruction groups fall into three major categories:

1. The groups 07600X - 07657X are for instructions which will be of general use across the range of PDP11 processors. The opcodes in this range will be characterized as (a) uniquely and immutably defined and (b) reasonable for implementation on all processor models of the PDP11 family.

2. The groups 07660X - 07667X are for instructions which will be used only on specific processors of the PDP11 family. These too will be uniquely and immutably defined, but each opcode will be restrictively assigned to a specific processor model and may not be implemented on other processors.

3. The groups 07670X - 07677X will neither be uniquely nor immutably defined but will be left available for free and indiscriminate customer usage.

2.7 OPERANDS FOR EXTENDED INSTRUCTIONS

Operands for extended instructions may be implicitly or explicitly specified. Explicit operands are specified, either in a general or in an opcode specific manner, through information expressed directly in the instruction stream. R7 is conceptually incremented by two as each word which contains an operand-specifier or operand in the instruction stream is fetched (-- see Section 2.7.3.4).

Implicitly specified operands do not appear in the instruction stream. If an instruction utilizes an implictly specified operand, the definition of that instruction will specify the exact location and format of such an operand.

2.7.1 Implicit Operands

Implicitly specified operands may be defined to be located:

1. in the general-purpose registers,

2. in defined machine registers,

3. on the R6 stack,

4. in defined locations in the virtual address space, or

5. in defined locations in the physical address space.

2.7.2 Explicit Opcode-Specific Operands

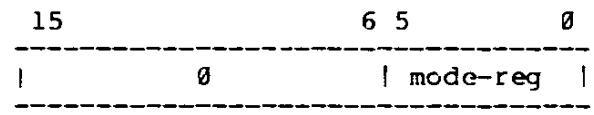
The definition of an instruction may specify that operands immediately follow it in the instruction stream. The format and interpretation of such operands can be specified in an opcode specific manner and will so be defined in the description of the instruction.

2.7.3 Explicit General Operands

2.7.3.1 Number and Types of General Operand Formats - When an instruction utilizes explicit general operand specifiers, the operand specifiers shall immediately follow the extended opcode in the instruction stream. As many operand specifiers as the instruction requires follow in consecutive order.

Instructions which utilize a single general operand will use the single-operand-specifier format (-- see Section 2.7.3.2). Instructions which require two consecutive explicit operands will use the double-operand-specifier format (-- see Section 2.7.3.3). Instructions which use more than two consecutive explicit operands will specify the operands in a succession of double-operand specifiers, and the last operand, when there are an odd number of operands, will be specified in the single-operand format.

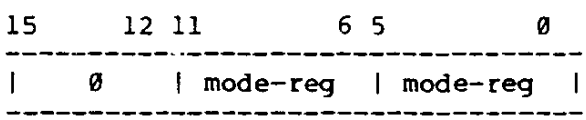
2.7.3.2 Format of Single-Operand Specifiers - The single-operand specifier consists of a word in the following format:



Bits <15:6> must be 0. Else a trap through vector 4(8) (invalid instruction specifier) will be taken.

Bits <5:0> specify the operand in the traditional PDP11 mode-register format.

2.7.3.3 Format of Double-Operand Specifiers - The double-operand specifier consists of a word in the following format:



Bits <15:12> must be 0. Else a trap through vector 4(8) (invalid instruction specifier) will be taken.

Bits <11:6> specify the first of the two operands, and bits <5:0> specify the second. Each operand is specified in the traditional PDP11 mode-register format.

2.7.3.4 Additional Instruction Stream Operand Words - For as many general operand specifiers as utilize mode 6 or 7 (with any register) or as utilize modes 2 or 3 with register 7, additional operand words are required in the instruction stream. These additional operand words immediately follow the operand-specifier word which calls for them.

Thus, for example, a hypothetical instruction:

```
ZAP   #A, (R1)+, B(R4), C, D
```

requiring explicit general operands would appear in the instruction stream as the following eight words:

opcode zzz for ZAP	076zzz
specifiers for operands 1 & 2	002721
value of literal A	aaaaaa
specifiers for operands 3 & 4	006467
value of index B	bbbbbb
displacement off PC for address of C	cccccc
specifier for operand 5	000067
displacement off PC for address of D	dddddd

2.8 TRAPS

2.8.1 Reserved-Instruction Traps -- Vector 10(8)

When an instruction is fetched which has a reserved or an unimplemented opcode, the processor shall trap through vector 10(8). The program counter (PC) contents which are stored on the kernel stack shall be the address of the word immediately following the trapped opcode word (i.e. old PC + 2), the processor status (PS) contents which are stored on the kernel stack shall contain the condition codes which represent the machine state immediately before the instruction was fetched. If the trap occurs in other than kernel mode, that mode's R6 is unchanged; if the trap occurs while executing in kernel mode, the kernel mode R6 will be 4 lower than its previous value. All other processor state (i.e. R0 through R5 of the selected general register set and memory) shall be exactly the same as it was when the trapped opcode was fetched.

On a multi-mode machine, some instructions may only be executed in kernel mode. If an attempt is made to execute them in a less privileged mode, a trap through vector 10(8) (reserved instruction) is to be taken. The processor state is preserved as stated above.

2.8.2 Trace Traps -- Vector 14(8)

T-bit traps are eligible for servicing only between instructions. Suspendable instructions as described in Section 2.9 will neither interfere with the servicing of T-bit traps nor stimulate T-bit traps during their execution.

2.8.3 Fatal Traps -- Vector 4(8)

Fatal conditions encountered in attempting to execute an instruction shall result, unless otherwise specified, in a trap through vector 4(8). When fatal traps occur, the processor state may not be the same as it was when the instruction was fetched, and the PC-address which is stored on the stack has no predictable relation to the address of the opcode word of the aborted instruction.

Events which result in a trap through vector 4(8) will set a bit in the CPU error register (if implemented) to indicate the condition which caused the trap. The CPU error register bits and conditions are:

0 illegal interrupt address access
1 USC parity error
2 red zone stack limit abort
3 yellow zone stack limit trap
4 bus time-out
5 non-existent memory
6 odd address error
7 illegal halt or micro break
8 invalid instruction specifier

Odd address error checking should be enabled to detect errors which may occur from the intermediate state of suspendable instructions.

Stack limit violations will refer to the furthest extent of the stack (or temporary data) during instruction execution. If the stack extends into the YELLOW zone during kernel mode execution, an internal interrupt request is generated. This will be handled in a similar way as externally generated interrupt requests. If the stack extends into the RED zone during kernel mode execution, the instruction is aborted, R6 is set to a value of 4, and a trap through vector 4(8) is taken. Note that RED zone aborts superceed YELLOW zone traps.

2.8.4 Floating Point Traps — Vector 244(8)

If a floating point processor is not present, all instructions in the floating point opcode space (17XXXX(8)) trap as reserved instructions through vector 10(8). If a floating point processor is present, illegal instructions in the floating point opcode space (170006(8), 170010(8) and 170013(8)-170077(8)) asynchronously trap through vector 244(8). Refer to the description of the floating point processor in the PDP11 Processor Handbook for additional information.

2.8.5 Other Traditional Traps

Other cases of traps (memory parity errors, memory management aborts, etc) are to be handled in the traditional PDP11 style (-- see Section 2.10).

2.8.6 Traps Unique to Extended Instructions

Traps required by extended instructions (e.g. invalid pointer, data exception, etc.) must not conflict with existing trap assignments and must be explicitly specified in the definition of the instruction in Chapter 5.

2.9 SUSPENDABLE INSTRUCTIONS

The intent of defining instruction suspendability is to establish a means for providing reasonable interrupt latency and does not presume to endow extended instructions with an ability to recover from trap conditions from which sequences of basic instructions cannot recover.

Suspension-events refer primarily to events which occur asynchronously to the instruction's execution; these are specifically the interrupts generated by I/O peripheral devices, power-fail traps, and floating point processor exceptions. Secondarily, suspension-events can refer also to those synchronous trap events which occur only for information notification purposes and do not imply that the integrity of the instruction's execution is in jeopardy. Such suspension events include YELLOW zone traps.

2.9.1 Suspendability Classifications

Each extended instruction is classified either as "non-suspendable" or as "potentially suspendable".

As explained below, two implementation choices are possible for non-suspendable instructions, and three are possible for potentially suspendable instructions. The following diagram can serve as a guide to subsequent portions of this section.

architecture	implementation
A) Non-Suspendable	1) non-interruptible 2) restartable
B) Potentially Suspendable	1) non-interruptible 2) restartable 3) suspendable

2.9.2 Non-Suspendable Instructions

A "non-suspendable" instruction has no architectural mechanism to allow it to be suspended while a suspension-event is serviced and then subsequently to be resumed.

A "non-suspendable" instruction may be implemented either as "non-interruptible" or as "restartable".

If an instruction is implemented as "non-interruptible", then once its execution has commenced, the processor will defer service of all suspension-events until after the completion of the instruction.

If an instruction is implemented as "restartable", then the instruction may be aborted to allow the processor to service suspension-events. The programmer visible state will be restored to that which existed immediately prior to the instruction execution. Upon the processor's return from servicing the suspension-event, the instruction will be started afresh.

2.9.3 Potentially Suspendable Instructions

"Potentially suspendable" instructions have a defined architectural mechanism, viz. PS<8> as described below, by which they can be suspended in mid-execution to allow the processor to service suspension-events and then subsequently to be resumed from the point where they had been suspended.

A "potentially suspendable" instruction may be implemented either as "non-interruptible" (-- see Section 2.9.2), as "restartable" (-- see Section 2.9.2) or as "suspendable" (-- see below).

The presence of suspension events may cause certain extended instructions to be suspended on some processors. If the instruction is suspended, PS<8> will be set, R7 will be backed up to address the opcode word, and the suspension event trap will be taken. When the instruction is resumed, PS<8> indicates that execution of the instruction has previously begun.

In order to make these instructions suspendable on all processors, the instruction state is part of the user state which is saved by interrupt handling routines. This includes the general registers, condition codes and memory. This state is processor dependent when suspended. Software should not attempt to interpret or modify this state; it must only be saved and restored. Up to 64(10) words of internal instruction state may also have been pushed onto the stack (-- see Section 2.10). This state must not be modified by software. The instruction will remove this state from the stack when it is resumed.

If PS<8> is set prior to executing a "potentially suspendable" instruction, the effect of the instruction is unpredictable (-- see Section 2.11).

At the normal completion of an "potentially suspendable" instruction, PS<8> will be cleared.

In order to promote uniform nomenclature, the name of the bit PS<8> will be "Instruction Suspension" with the corresponding mnemonic "IS".

2.9.4 Instruction Suspension

All suspendable instructions will use PS<8> to indicate instruction suspension. When a potentially suspendable instruction is executed, PS<8> cleared means that the instruction is being commenced; set means that the instruction is being resumed. It will be cleared upon successful completion of any suspended instruction. PS<8> will be cleared when:

1. A suspended instruction successfully completes.
2. Processor power-up.
3. New PS is fetched from vector location with PS<8> clear.
4. RTI or RTT is executed with new PS<8> clear.
5. PS<8> explicitly cleared by an instruction.

PS<8> will be set when:

1. Potentially suspendable instruction is interrupted and wishes to be suspended.

2. New PS is fetched from vector location with PS<8> set.

3. RTI or RTT is executed with PS<8> set.

4. PS<8> is explicitly set by an instruction.

The setting of this bit will have no affect on instructions which are not potentially suspendable; such instructions will not implicitly modify this bit.

When an instruction is suspended the following state may contain information vital to the resumption of the instruction. This information must be preserved, and restored prior to restarting the suspended instruction. This information is processor model dependent; it may vary from one execution of the instruction to another.

1. General registers R0 through R5.

2. Condition code bits (PS<3:0>).

3. Up to 64(10) words on the stack of the context in which the suspended instruction was executing.

4. Any destinations used by the instruction.

2.10 STACK UTILIZATION

Extended instructions may use the R6 stack for temporary "scratch" state storage.

The maximum number of additional words which an extended instruction may claim on the R6 stack is 64(10). The reason for imposing a limit is to ensure that system software can adequately provide for worst-case stack allocation requirements. In addition to the above restriction, the normal PDP11 stack-limit mechanism remains in effect for extended instructions just as it does for any other instruction.

If an extended instruction is interrupted, R6 must have been updated to encompass any additional stack storage still required for completion of the instruction.

All extended instructions will support dyanmic stack allocation facilities used by some software systems. This means that memory management traps which result from over-extending the stack area must be survivable. If insufficient stack space exists, the instruction must terminate by a memory management abort in such a way that if additional stack space were allocated, the instruction could be successfully restarted.

2.11 UNPREDICTABLE CONDITIONS

"Unpredictable" means that the outcome is indeterminate and non-repeatable. Either the results of an instruction or the effect of an instruction can be unpredictable. When the results of an instruction are unpredictable, the condition codes and destination operands (but not their descriptors) will contain unpredictable values; destinations may not even contain valid results. When the effect of an instruction is unpredictable, the entire user or process state, and not only the portion typically used by the instruction will be unpredictable. In a machine with multiple modes and address spaces, an unpredictable operation in a less privileged mode will not affect the state of a more privileged mode, nor will it result in accesses to memory from user mode which are outside the mapped limits of the user's program.

Note that architectural constraints exist on unpredictable effects. In particular, an unpredictable effect which manifests itself as a trap must meet all the requirements for the particular trap (-- see Section 2.8).

Implementors are encouraged to select the manifestations of unpredictable results and effects to be such that their occurrence is visible to software at the earliest possible time.

2.12 RESERVATION OF UNUSED FIELDS AND ENCODINGS

Fields and encodings which are available to an instruction, but are not used, are reserved by the architecture. This will permit future definition of these not to conflict with existing software.

Any unused field (single bit or contiguous group of bits) must be zero if it is reserved by the architecture. Any non-zero value in the field will cause the effect of such an instruction to be unpredictable.

Any unused encoding (field of n bits where less than 2^n encodings are defined) is reserved by the architecture. Use of such encodings will cause the effect of such an instruction to be unpredictable.

2.13 MULTIPROGRAMMING INTEGRITY

Machine implementations shall ensure that, under all initial settings of registers and memory, extended instructions shall not violate any bound implicit in multiprogrammed operation. Specifically, the following are to be avoided:

1. A less-privileged program escaping into a higher-privileged mode.

2. A program escaping beyond its address-mapping limits.

3. A non-interruptable or non-terminating sequence.

4. Excessive interrupt latency.

CHAPTER 3

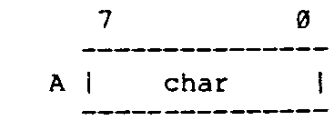
Extended-Instruction Data Types

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:

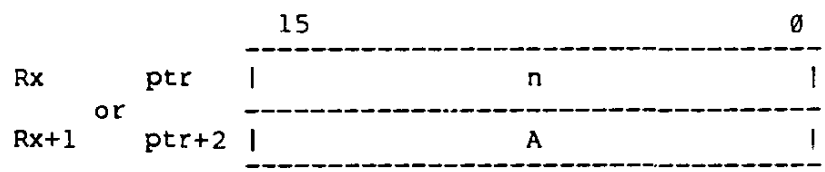


The character is used as an operand by CIS11 instructions. When it appears in a general register, the character is in the low order half; the high order half of the register must be zero. When it appears in the instruction-stream, the character is in the low order half of a word; the high order half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

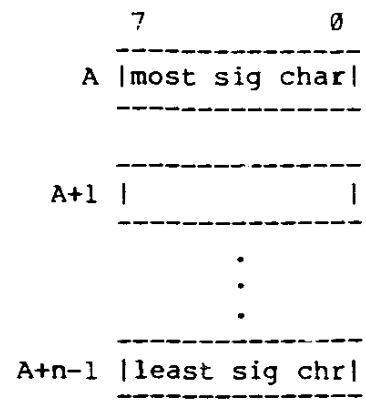
3.1.2 Character String

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:



The following figure shows the character string in memory:

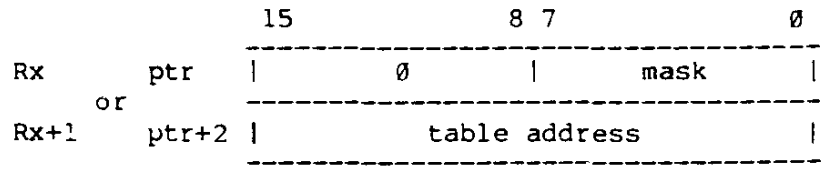


3.1.3 Character Set

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

Operationally, a character (char) is considered to be in the character set if the evaluation of (M[table.adr+char] AND mask) is not equal to zero. The character is not in the character set if the evaluation is zero. Each byte in the table indicates which combination of up to eight orthogonal character subsets (i.e. one for each of the eight bit vectors 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.



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 (-- see section 2.11) if a decimal string used as a source operand contains an invalid digit encoding, invalid sign designator, or in the case of overpunched numbers, an invalid sign/digit encoding.

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

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

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

3.2.2 Decimal String Descriptors

Decimal strings are represented by a two word descriptor. The descriptor contains the length, data type, and address of the string. It appears in two consecutive general registers (register form of instructions), or in two consecutive words in memory pointed to by a word in the instruction stream (in-line form of instructions). The unused bits are reserved by the architecture and must be Ø. The effect of an instruction using a descriptor will be unpredictable if any non-zero reserved fields in the descriptor contain non-zero values or a reserved data type encoding is used (-- see sections 2.11 and 2.12). The design of the numeric and packed string descriptors are identical:

First Word:

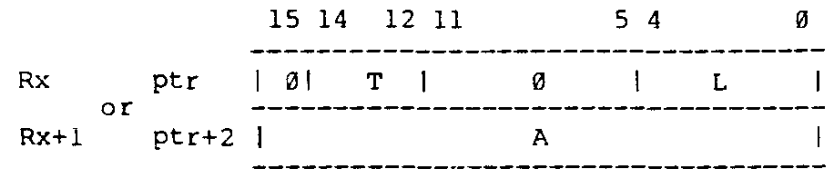
length <4:Ø> - Number of digits specified as an unsigned binary integer.

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

Second Word:

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

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



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

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

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

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

3.2.3 Packed Strings

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

Signed Packed Strings -

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

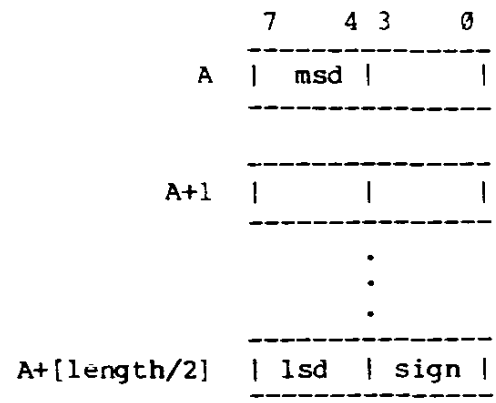
Unsigned Packed Strings -

PACKED SIGN NIBBLE:

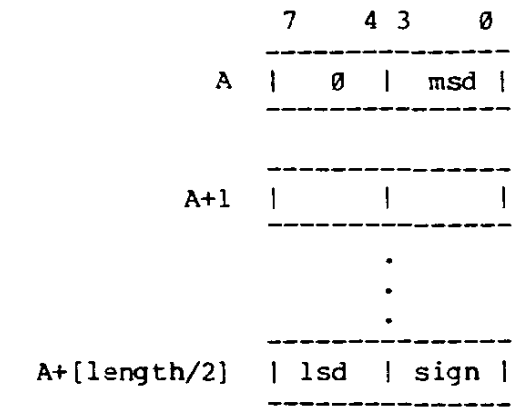
Sign Nibble	Preferred Designator	Alternate Designators
positive	1100(2)	1010(2) 1110(2) 1111(2)
negative	1101(2)	1011(2)
unsigned	1111(2)	

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

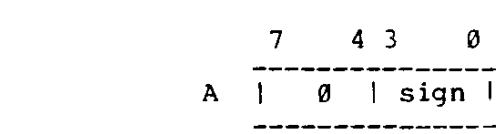
The following is a packed string with an odd number of digits:



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



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



A valid packed string is characterized by:

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

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

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

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

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

3.2.4 Zoned Strings

<7:4>) and the low order nibble (bits <3:0>). The low order nibble contains the value of the corresponding decimal digit.

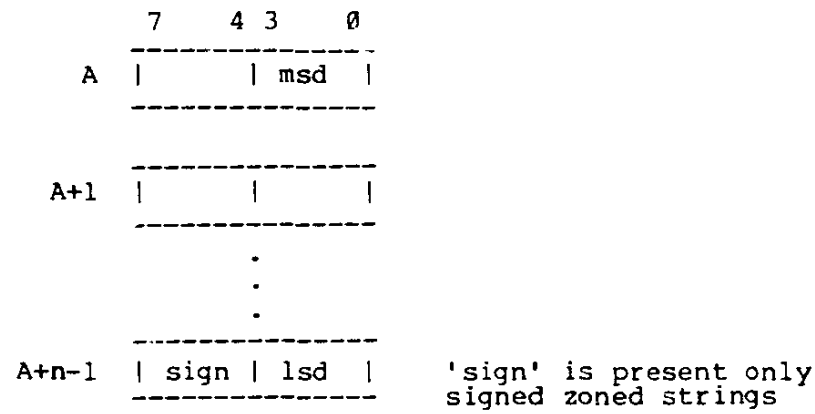
Signed Zoned Strings -

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

Unsigned Zoned Strings -

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

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



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

A valid zoned string is characterized by:

1. A length from 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.

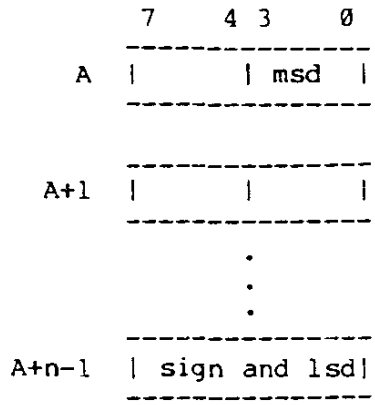
The following table shows the sign of the decimal string and the value of the digit which is encoded in the sign byte. Source strings will properly accept both the preferred and alternate designators; destination strings will store the preferred designator. The preferred designators correspond to the ASCII graphics 'A' to 'R', '{' and '}'. The alternate designators correspond to the ASCII graphics '0' to '9', '[', '?', ']', '!' and ':'.

OVERPUNCH SIGN/DIGIT BYTE:

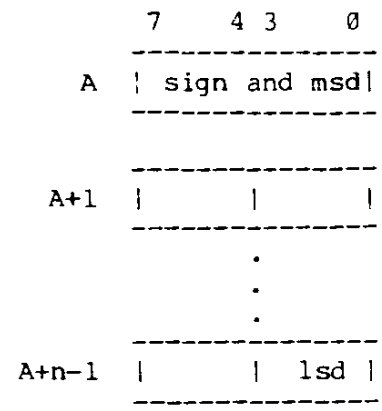
Overpunch Sign/Digit	Preferred Designator	Alternate Designators
+0	01111011(2)	00110000(2), 01011011(2), 00111111(2)
+1	01000001(2)	00110001(2)
+2	01000010(2)	00110010(2)
+3	01000011(2)	00110011(2)
+4	01000100(2)	00110100(2)
+5	01000101(2)	00110101(2)
+6	01000110(2)	00110110(2)
+7	01000111(2)	00110111(2)
+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:



The following is a leading overpunch string:



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

A valid overpunch string is characterized by:

1. A length from 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.

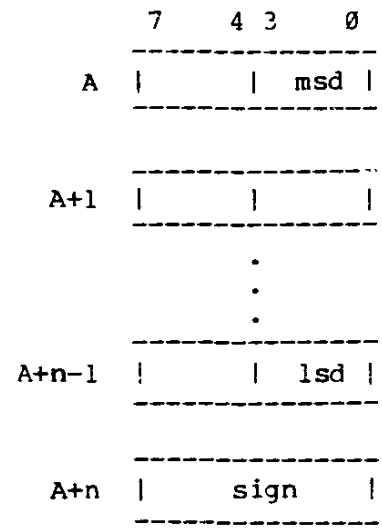
When used as a source string the high order nibbles of all digit bytes are ignored. Destination strings are stored with 0011(2) in the high order nibble of all digit bytes. 0011(2) in the high order nibble corresponds to the ASCII encoding for numeric digits. The preferred positive sign designator is 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:

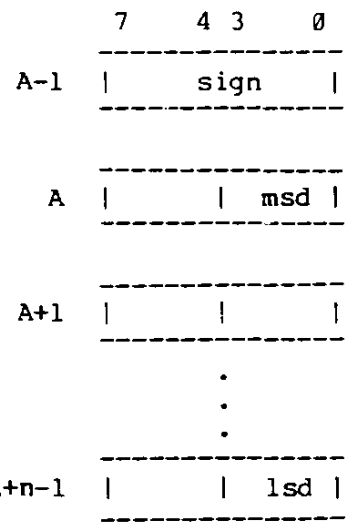
Sign Byte	Preferred Designator	Alternate Designators
positive	00101011(2)	00100000(2)
negative	00101101(2)	

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:

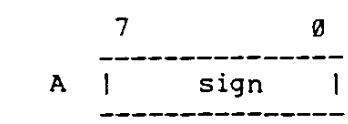


The following is a leading separate string:

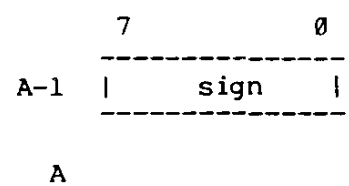


A zero length separate string occupies a single byte of memory which contains the sign. When used as a source, zero length strings provide operands with zero magnitude; when used as a destination, they can only reflect a result of zero magnitude without indicating overflow; the sign of the result is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

The following is a zero length trailing separate string:



The following is a zero length leading separate string:



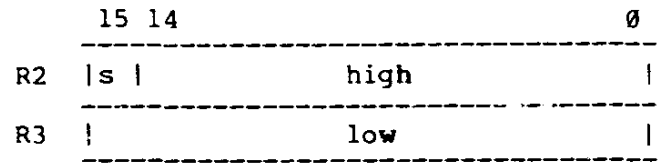
A valid separate string is characterized by:

1. A length from 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).

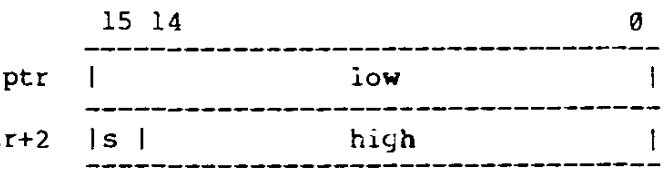
3.3 LONG INTEGER

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:



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



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

CHAPTER 4

Description and Intent of Extended Instructions

4.1 PROCESSOR IDENTIFICATION INSTRUCTION

The MFPT instruction provides software with a means of identifying the processor model on which it is executing. The instruction returns a word in R0 whose low order half is an 8 bit processor model code and whose high order half is an 8 bit processor subcode. The processor model code is uniquely assigned for each processor type which implements the instruction. A processor type is differentiated by having its own option designation (e.g., KD11-x). The processor subcode may be used in any meaningful way by the processor implementor. Possible uses would be to indicate the micro-code or module revision, configuration options which are present, etc. This instruction will be incorporated in the basic instruction set of all PDP11 processors (-- see section 1.4.3). The processor model codes will be assigned by the PDP11 Architecture Group Manager.

4.2 COMMERCIAL INSTRUCTION SET

The PDP11 Commercial Instruction Set (CIS11) consists of the following extended instruction groups:

07602X	Commercial Load 2 Descriptors
07603X	Character String Move
07604X	Character String Search
07605X	Numeric String
07606X	Commercial Load 3 Descriptors
07607X	Packed String
07613X	Character String Move (in-line)
07614X	Character String Search (in-line)
07615X	Numeric String (in-line)
07617X	Packed String (in-line)

These include instructions which operate on character strings and on decimal numbers. Each generic type of instruction is provided in two forms. The essential difference between the two forms is the manner in which operands are delivered to the instruction. The first form is the 'register form' where operands are implicitly obtained from the general registers. The second form is the 'in-line' form where operands or word address pointers to operands follow the opcode word in the instruction stream. The mnemonic for the in-line form is the mnemonic for the register form suffixed with the letter 'I'. The condition codes are set identically for both forms. The in-line forms minimize register modification.

Instructions are also provided which efficiently load operands into the general registers.

4.2.1 Character String Instructions

The character string operations conveniently provide most of the common, as well as time consuming functions that are encountered in commercial data and text processing applications.

4.2.1.1 Instructions - Instructions are provided to move and to search character strings:

Character String Move Instructions	
MOVC(I)	move character
MOVRC(I)	move reverse justified character
MOVTC(I)	move translated character

Character String Search Instructions	
LOCC(I)	locate character
SKPC(I)	skip character
SCANC(I)	scan character
SPANC(I)	span character
CMPC(I)	compare character
MATC(I)	match character

4.2.1.1.1 Character String Move Instructions - The character string move instructions use character string descriptors as operands. These descriptors specify a source and a destination character string. The contents of the source are moved to the destination with alignment at either the most significant character as in MOVC(I) and MOVTC(I), or the least significant character as in MOVRC(I). If the source is longer than the destination, characters are truncated from the side opposite that of the alignment; if the destination is longer than the source, the destination is completed with fill characters on the side opposite that of the alignment. The MOVTC(I) instructions move a translated source string to a destination string.

4.2.1.1.2 Character String Search Instructions - The character string search instructions use a character string descriptor as one operand. The other operand is either a character, a character string descriptor, or a character set descriptor. These instructions are used to examine the source string to find the presence or absence of characters. The source string is processed from most significant to least significant character.

Conceptually, these instructions may be divided into 3 classes:

1. Character String Searches - CMPC(I) compares two character strings. The condition codes are set according to the comparison of the corresponding most significant unequal characters. MATC(I) finds an object string within a source string. This is the 'instring' function that languages and text processing systems provide.

2. Character Searches - LOCC(I) finds the first occurrence of a given character in a string. SKPC(I) skips to the first non-occurrence of a given character in a string.

3. Character Set Searches - In these instructions, a string is examined until a member of a character set is either found as in SCANC(I), or not found as in SPANC(I). This aids the search for one of several delimiters such as '/', ',', CR, LF, FF, etc, or the passing of combinations of characters such as blanks, tabs, etc. LOCC(I) and SKPC(I) are optimizations of SCANC(I) and SPANC(I) in which the set consists of a single character.

4.2.1.2 Condition Codes - The setting of condition codes reflects the result of the character string operations. For character string moves, the condition codes indicate whether the source and destination strings were of equal length, the source was shorter than the destination such that fill characters were used, or the source was longer than the destination such that characters were truncated. This is accomplished by setting the condition codes on the result of arithmetically comparing the initial source and destination lengths. For CMPC(I), the condition codes are the result of arithmetically comparing the most significant corresponding pair of unequal characters. For the other search instructions, they show whether or not the operand strings were completely examined.

The condition codes for some character string search instructions may be interpreted according to the notion of success or failure. Success is the accomplishment of the instruction's task; failure is the inability to accomplish the task. Since the condition codes are set based on the results of the instruction, there is an indirect correspondence between these settings and success or failure. This correspondence is invariant within an instruction, but it is not the same for all search instructions. Therefore, different branch instructions must be used to test the operation of each instruction. They are summarized in the following table:

Instruction	Success	Failure
LOCC(I)	BNE	BEQ
SCANC(I)	BNE	BEQ
CMPC(I)	BEQ	BNE
MATC(I)	BNE	BEQ

4.2.1.3 Operand Delivery - The "register form" of character string instructions implicitly find operands in the general registers. These operands include character, character string descriptor, character set descriptor, and translation table address. If an instruction does not use a register, its contents will be undisturbed. RØ-R1 generally contain a source character string descriptor; R2-R3 generally contain a second source character string descriptor, or the destination string descriptor. The low order half of R4 is used as an explicit character. R4-R5 is used to contain a character set descriptor. R5 contains the starting address of a 256 byte table which is used for character translation.

When move instructions terminate, RØ contains the number of unmoved source characters, and R1, R2, and R3 are cleared. For search instructions, the registers are updated to represent descriptors for the resulting strings.

The "in-line form" of character string instructions find operands, or pointers to operands, in the instruction stream immediately following the opcode word. Operands which appear directly in the instruction stream include characters, and translation table addresses. Descriptors are represented in the instruction stream by a single word whose contents are interpreted as a word address pointer to the two word descriptor. These descriptors specify character strings and character sets. Some instructions return a character string descriptor in RØ-R1.

4.2.1.4 Data Overlap - In general, all character string instructions are unaffected by the overlapping of source or destination strings. The result of the move instructions is equivalent to having read the entire source string before storing characters in the destination. If the destination string of the MOVTC(I) instructions overlaps the translation table, the characters stored in the destination string will be unpredictable.

4.2.1.5 Unpredictable Conditions - The effect of character string instructions will be unpredictable if:

1. PS<8> is non-zero when the instruction is first started -- this bit contains suspension information.

2. R6<0> is non-zero when the instruction is first started -- the stack pointer must contain a word address.

3. R6 does not contain the address of a 64(10) word stack -- temporary state for instructions.

4. Bits <15:8> of the word containing a character operand is non-zero.

5. Bits <15:8> of the first word of a character set descriptor is non-zero.

6. A source string overlaps the 64(10) word stack or I/O page.

7. A destination string overlaps the opcode word, in-line operands, 64(10) word stack, I/O page, or trap vectors.

8. A table overlaps the stack or I/O page.

9. The opcode word or in-line operands overlap the destination string, 64(10) word stack or I/O page.

10. The stack overlaps the source string, destination string, table, opcode word, in-line operands, I/O page, or trap vectors.

11. The I/O page overlaps the source string, destination string, table, opcode word, in-line operands or 64(10) word stack.

Character string instructions will produce unpredictable results if:

1. MOVTC(I) -- Destination string overlaps the translation table.

2. MOVTC(I), SCANC(I), SPANC(I) -- The entire 256 byte translation or character set tables are not in readable memory.

4.2.1.6 Implementation Notes -

1. Source character strings, opcodes, words in the instruction stream, and descriptors for in-line instructions must be in readable memory; they need not be in writable memory. Destination strings must be in memory which is both readable and writable. Stacks must be in memory which is both readable and writable.

2. Neither the order, width, number nor type of operand accesses is architected.

3. On machines with multiple register sets, these instructions will use the register set selected by PS<11>.

4. On machines with multiple modes, these instructions will use the stack pointer and memory map selected by PS<15:14>.

5. On machines with I and D memory spaces, the I space will be used for instruction stream fetches (opcodes, in-line operands and in-line pointers), and the D space will be used for descriptors (in-line instruction form) and data references.

6. For vacant strings, the address must not be used and no memory references are to be made.

7. Instructions can use as much as 64(10) words on the stack. This stack space can be used whether the instruction is suspended or not; it is however exclusive of the PC and PS which is pushed on the stack if the instruction is suspended. When instructions terminate normally, R6 will be restored to its original value, but the contents of the 64(10) words immediately below the stack are unpredictable.

8. If word pointers contain an odd address, set CPU Error Register<6> and then trap through vector 4(8).

4.2.2 Decimal String Instructions

The decimal string instruction groups aid manipulation of decimal data. Several numeric (byte) and packed decimal data types are supported. Instructions are provided for basic arithmetic operations, as well as for compare, shift, and convert functions.

4.2.2.1 Instructions - Each arithmetic, shift and compare instruction operates on a single class of data type. Both numeric and packed string instructions are provided for most operations. Convert instructions have a source operand of one data type and a destination operand of another data type. Decimal string instructions specify to which class each of their decimal string operands belong. The data type supplied as part of each operand's descriptor may be any valid data type of the class. This permits a general mixing of data types within each of numeric and packed classes.

The data types on which an instruction operates are designated by the last letter(s) of the opcode mnemonic. 'N' denotes numeric strings, 'P' denotes packed strings, and 'L' denotes long binary integers.

The arithmetic instructions are ADDN(I), ADDP(I), SUBN(I), SUBP(I), MULP(I) and DIVP(I). ASHN(I) and ASHP(I) shift a decimal string by a specified number of digit positions (either direction) with optional rounding and store the result in the destination string. Thus, they effectively multiply or divide by a power of ten. If the shift count is zero, these shift instructions can be used simply to move decimal strings (destinations are stored with preferred representation). Move negated may be accomplished by using SUBN(I) or SUBP(I). Arithmetic comparison instructions, CMPN(I) and CMPP(I), are provided to examine the relative difference between two decimal strings.

CVTNL(I) and DVTPL(I) convert a decimal string to a long (32 bit) two's complement integer. CVTLN(I) and CVTLP(I) convert a long integer to a decimal string. CVTNP(I) and CVTPN(I) convert between numeric and packed decimal strings.

The instructions are:

Numeric String Instructions	
ADDN(I)	add numeric
SUBN(I)	subtract numeric
ASHN(I)	arithmetic shift numeric
CMPN(I)	compare numeric

Packed String Instructions	
ADDP(I)	add packed
SUBP(I)	subtract packed
MULP(I)	mulitply packed
DIVP(I)	divide packed
ASHP(I)	arithmetic shift packed
CMPP(I)	compare packed

Convert Instructions	
CVTNL	convert numeric to long
CVTLN	convert long to numeric
CVTPL	convert packed to long
CVTLP	convert long to packed
CVTNP	convert numeric to packed
CVTPN	convert packed to numeric

4.2.2.2 Condition Codes - For instructions which store a value in a destination string, the N and Z bits reflect the value stored. The N bit indicates a negative destination; the Z bit indicates a destination having zero magnitude. A destination string with zero magnitude is considered to be positive (even if a negative zero was stored as a consequence of decimal overflow). Thus, the setting of N and Z are mutually exclusive.

The V bit will indicate whether the destination string accurately represents the true result of the instruction. It is also set if a division by zero was attempted. If the V bit is set, the destination string will represent the least significant portion of the result (truncated). If the V bit is cleared, the destination represents the true result.

For DIVP(I), C indicates division by zero. Otherwise, C is always cleared.

For comparisons using the CMPN(I) and CMPP(I) instructions, the N and Z bits reflect the signed relationship between the source strings. The signed branch instructions can test the result. V and C are cleared.

For instructions which return a long integer value, N reflects the sign of the two's complement integer, and Z indicates whether it was zero. V indicates whether the long integer could not contain all significant digits and sign of the result. CVTNL(I) and CVTPL(I) also use C to represent a borrow from a more significant portion of the long binary result. Otherwise, C is cleared.

4.2.2.3 Operand Delivery - The "register form" of decimal string instructions implicitly find their operands in the general registers. These operands include decimal string descriptors, long binary integers, and shift descriptor words. If an instruction does not use a register, its contents will be undisturbed. R0-R1 generally contain the first source descriptor, R2-R3 generally contain the second source descriptor, and R4-R5 generally contain the destination descriptor. ASHN and ASHP use R4 to contain a shift descriptor word. CVTLN, CVTLP, CVTNL and CVTPL use R0-R1 to contain a decimal string descriptor, and R2-R3 for the long integer. When the instruction is completed, the source descriptor registers are cleared.

The "in-line form" of decimal string instructions find their operands, or pointers to descriptors in the instruction stream immediately following the opcode word. Operands which appear directly in the instruction stream are shift descriptor words. Operands which are represented in the instruction stream by a pointer containing the word address of the descriptor are decimal string descriptors and long binary integers. No in-line form of decimal string instructions modify R0-R6.

4.2.2.4 Data Overlap - The operation of decimal string instructions is unaffected by any overlap of the source operands provided that each source operand is a valid representation of the specified data type.

The overlap of the destination string and any of the source strings will, in general, produce unpredictable results. However, ADDN(I), ADDP(I), SUBN(I) and SUBP(I) will permit the destination string to overlap either or both source strings only if all corresponding digits of the strings are in coincident bytes in memory. This facilitates two address arithmetic.

4.2.2.5 Unpredictable Conditions - The effect of decimal string instructions will be unpredictable if:

1. PS<8> is non-zero when the instruction is first started -- this bit contains suspension information.

2. R6<Ø> is non-zero when the instruction is first started -- the stack pointer must contain a word address.

3. R6 does not contain the address of a 64(1Ø) word stack -- temporary state for instructions.

4. A source string overlaps the 64(1Ø) word stack or I/O page.

5. A destination string overlaps the opcode word, in-line operands, 64(1Ø) word stack, I/O page, or trap vectors.

6. The opcode word or in-line operands overlap the destination string, 64(1Ø) word stack or I/O page.

7. The stack overlaps the source string, destination string, table, opcode word, in-line operands, I/O page, or trap vectors.

8. The I/O page overlaps the source string, destination string, table, opcode word, in-line operands or 64(1Ø) word stack.

9. Bits <15> and <11:5> of the decimal string descriptors containing the string length are non-zero.

10. Reserved data type codes are used in bits <14:12> of decimal string descriptors.

11. ASHN(I)/ASHP(I) -- Bits <15:12> of the shift descriptor word are non-zero.

Decimal string instructions will produce unpredictable results if:

1. Source operands are not valid Decimal Strings.

2. Destination strings overlap source strings (except if all corresponding digits are coincident for ADDN(I), ADDP(I), SUBN(I) and SUBP(I)).

3. DIVP(I) -- Division by zero is attempted (only destination string, N and Z are unpredictable).

4. ASHN(I)/ASHP(I) -- Bits <11:8> of the shift descriptor word are 1010(2) to 1111(2).

5. CVTNP(I)/CVTPN(I) -- Source and destination strings overlap.

4.2.2.6 Implementation Notes -

1. Source decimal strings, opcodes, words in the instruction stream, descriptors and long integer sources for in-line instructions must be in readable memory; they need not be in writable memory. Destination strings and long integer destinations for in-line instructions must be in memory which is both readable and writable. Stacks must be in memory which is both readable and writable.

2. Neither the order, width, number nor type of operand accesses is architected.

3. On machines with multiple register sets, these instructions will use the register set selected by PS<11>.

4. On machines with multiple modes, these instructions will use the stack pointer and memory map selected by PS<15:14>.

5. On machines with I and D memory spaces, the I space will be used for instruction stream fetches (opcodes, in-line operands and in-line pointers), and the D space will be used for descriptors and long integers (in-line instruction form) and data references.

6. For zero length decimal strings of type signed zoned, unsigned zoned, leading overpunch and trailing overpunch, no memory is occupied. The address must not be used and no memory references are to be made.

7. Instructions can use as much as 64(10) words on the stack. This stack space can be used whether the instruction is suspended or not; it is however exclusive of the PC and PS which is pushed on the stack if the instruction is suspended. When instructions terminate normally, R6 will be restored to its original value, but the contents of the 64(10) words immediately below the stack are unpredictable.

8. If word pointers contain an odd address, set CPU Error Register<6> and then trap through vector 4(8).

4.2.3 Commercial Load Descriptor Instructions

The commercial load descriptor instructions augment the character and decimal string instructions by efficiently loading the general registers with string descriptors. Two forms of instructions are provided. The L2Dr instructions load two string descriptors into the general registers. The first descriptor is loaded into RØ-R1 and the second descriptor is loaded into R2-R3. This instruction supports the following:

equal length character string move
equal length character string compare
character string matching
decimal string compare

The second form, the L3Dr instructions, take three descriptors. The first is loaded into RØ-R1, the second into R2-R3, and the third into R4-R5. This instruction supports the following:

3-address arithmetic

The condition codes are not affected.

The descriptors are accessed by the following mechanism. Words containing the addresses of the descriptors (two for L2Dr and three for L3Dr) are in consecutive locations in memory. The descriptor addresses are found by applying the addressing mode @(Rr)+ once for each descriptor. The value of r is encoded as the low order three bits of the instruction's opcode. If Ø<=r<=5, then r can be thought of as the base address of a small table in memory, where each entry in the table contains the address of a descriptor. If r=6, then the instructions effectively pop the addresses of descriptors off of the stack. If r=7, then the descriptor addresses are contiguous with the instruction's opcode word.

The string descriptors are two words long. The address of the descriptor is that of the low order word. It is loaded into the corresponding even register. The high order word of the descriptor is loaded into the corresponding odd register. Note that although these instructions are described in terms of string descriptors, they are applicable for other instances where two consecutive words in memory referenced by a pointer are to be copied into even-odd general register pairs.

The instructions are:

Commercial Load Descriptor Instructions	
L2D0	load 2 descriptors using @(R0)+
L2D1	load 2 descriptors using @(R1)+
L2D2	load 2 descriptors using @(R2)+
L2D3	load 2 descriptors using @(R3)+
L2D4	load 2 descriptors using @(R4)+
L2D5	load 2 descriptors using @(R5)+
L2D6	load 2 descriptors using @(R6)+
L2D7	load 2 descriptors using @(R7)+
L3D0	load 3 descriptors using @(R0)+
L3D1	load 3 descriptors using @(R1)+
L3D2	load 3 descriptors using @(R2)+
L3D3	load 3 descriptors using @(R3)+
L3D4	load 3 descriptors using @(R4)+
L3D5	load 3 descriptors using @(R5)+
L3D6	load 3 descriptors using @(R6)+
L3D7	load 3 descriptors using @(R7)+

4.2.3.1 Implementation Notes -

1. Opcodes, words in the instruction stream, and descriptors must be in readable memory; they need not be in writable memory.

2. Neither the order, width, number nor type of operand accesses is architected.

3. On machines with multiple register sets, these instructions will use the register set selected by PS<11>.

4. On machines with multiple modes, these instructions will use the stack pointer and memory map selected by PS<15:14>.

5. On machines with I and D memory spaces, the I space will be used to fetch instructions as well as the descriptor addresses for L2D7 and L3D7; D space will be used to fetch descriptor address for L2D0-6, L3D0-6, and all string descriptors.

6. If word pointers contain an odd address, set CPU Error Register<6> and then trap through vector 4(8).

4.3 PROCESSOR SPECIFIC INSTRUCTIONS

The processor specific instructions provide model dependent diagnostic capability.

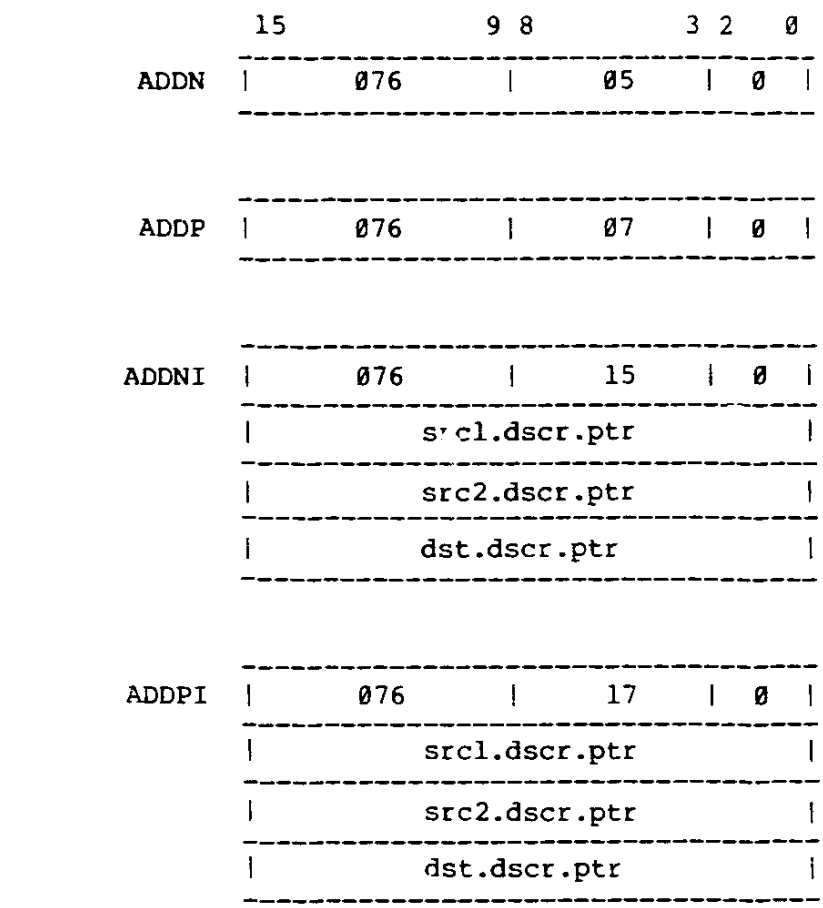
THIS IS A BLANK PAGE

CHAPTER 5

Extended-Instruction Definitions

5.1 ADDN / ADDP / ADDNI / ADDPI - Add Decimal

Format:



Operation:

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

Condition Codes:

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

Suspendability:

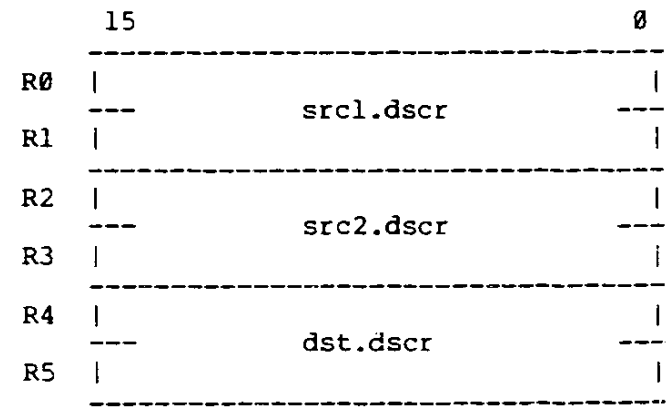
This instruction is potentially suspendable.

Description:

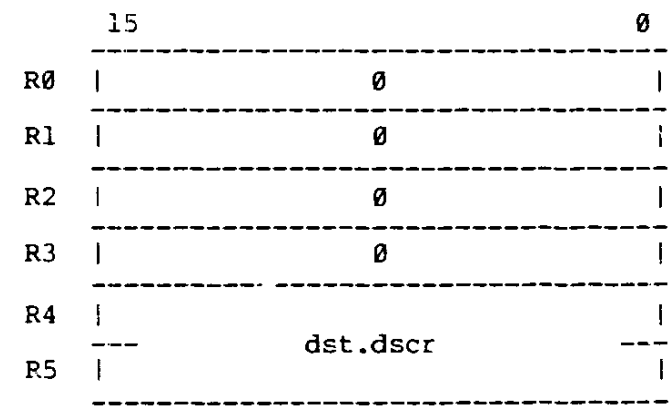
Src1 is added to src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

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:



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



In-line Form - ADDNI and ADDPI

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

Formal Description:

TBS;

Examples:

1. Three Address Add - Register Form

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

2. Three Address Add - In-line Form

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

3. Two Address Add - Register Form

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

4. Two Address Add - In-Line Form

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

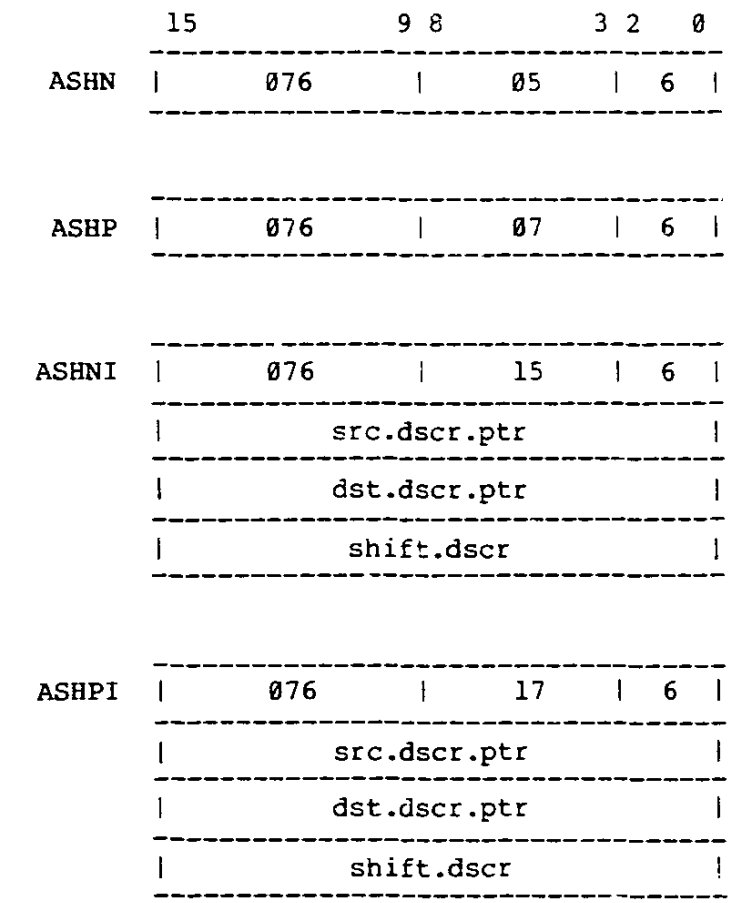
Notes:

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

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

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

Format:



Operation:

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

Condition Codes:

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

Suspendability:

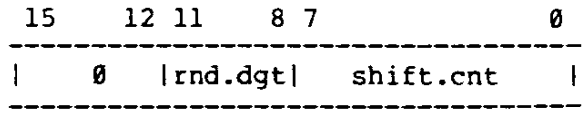
This instruction is potentially suspendable.

Description:

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

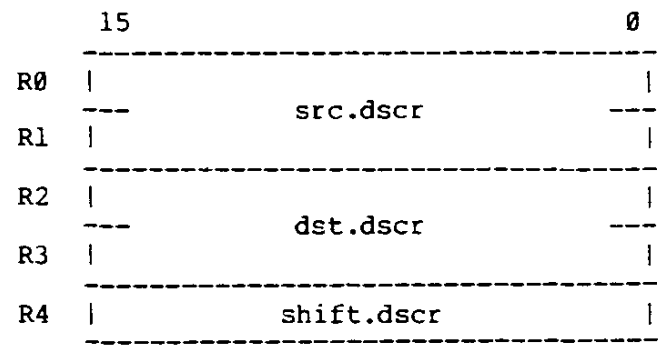
A negative shift count invokes a rounding operation. The result is constructed by shifting the source the specified number of digit positions. The rounding digit is then added to the most significant digit which was shifted out. If this sum is less than 10(10), the shifted result is stored in the destination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

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

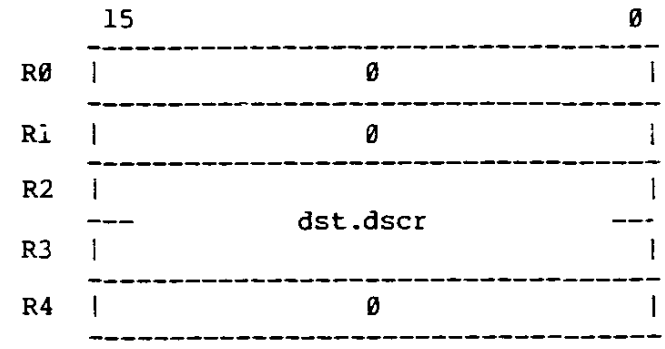


Register Form - ASHN and ASHP

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



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



In-line Form - ASHNI and ASHPI

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

Formal Description:

TBS;

Examples:

1. Multipling by 100 - Register Form

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

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

2. Multipling by 100 - In-line Form

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

3. Move decimal number - Register Form

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

4. Move decimal number - In-line Form

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

Notes:

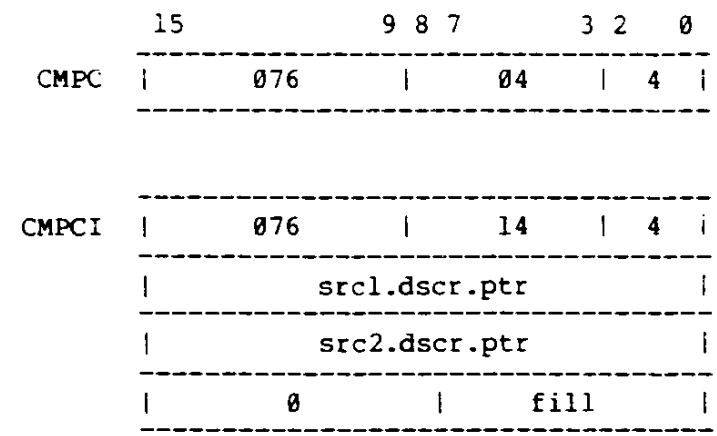
1. If bits <15:12> of the shift descriptor word are not zero, the effect of the instruction is unpredictable.

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

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

5.3 CMPC / CMPCI - Compare Character

Format:



Operation:

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.

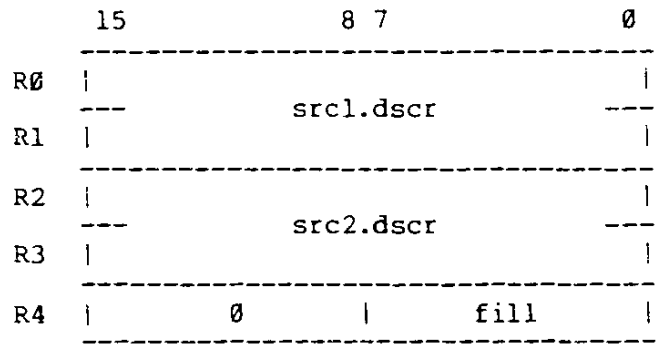
Description:

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.

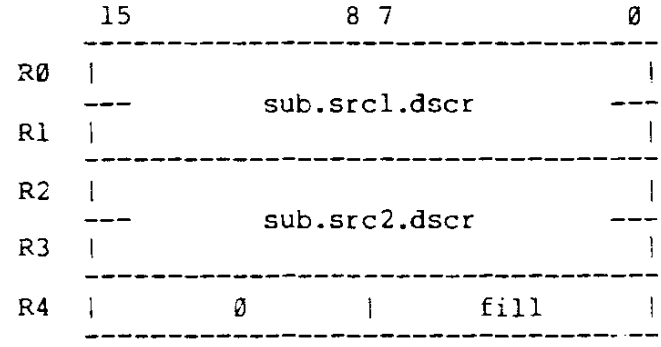
The condition codes reflect the last comparison, permitting the unsigned branch instructions to test the result.

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:



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.



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 fil character and whose high order half must be zero. R0-R6 are unchanged when the instruction is completed.

Formal Description:

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

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

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

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

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

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

Examples:

1. Compare Strings - Register Form

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

2. Compare Strings - In-line Form

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

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

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

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

Notes:

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

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

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

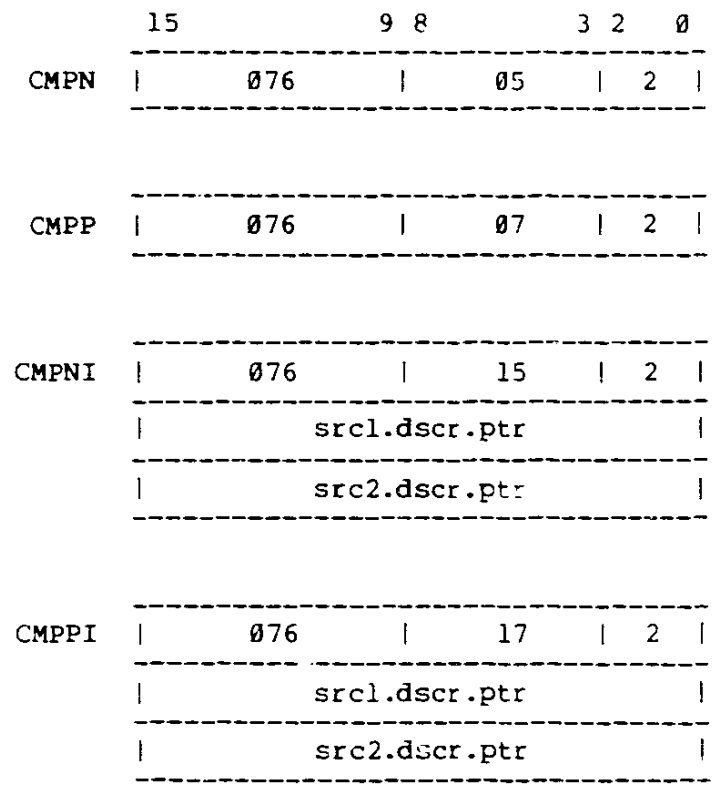
4. A test for success is BEQ; a test for failure is BNE.

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

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

5.4 CMPN / CMPP / CMPNI / CMPPI - Compare Decimal

Format:



Operation:

Src1 is compared with src2 (src1-src2).

Condition Codes:

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

Suspendability:

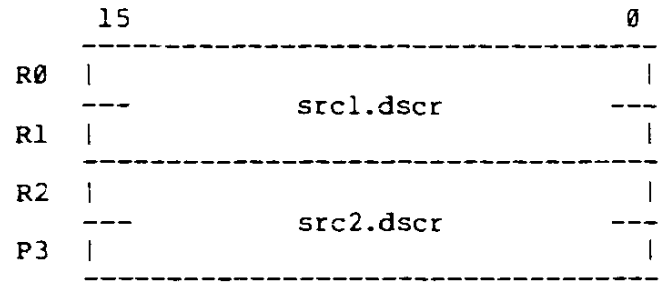
This instruction is potentially suspendable.

Description:

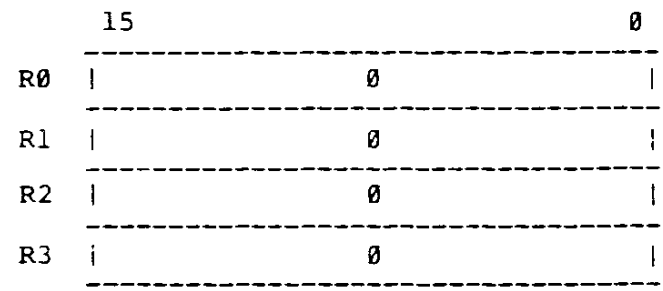
Src1 is arithmetically compared with src2. The condition codes reflect the comparison. The signed branch instruction can be used to test the result.

Register Form - CMPN and CMPP

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



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



In-line Form - CMPNI and CMPPI

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

Formal Description:

TBS;

Examples:

1. Compare Decimal Strings - Register Form

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

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

2. Compare Decimal Strings - In-line Form

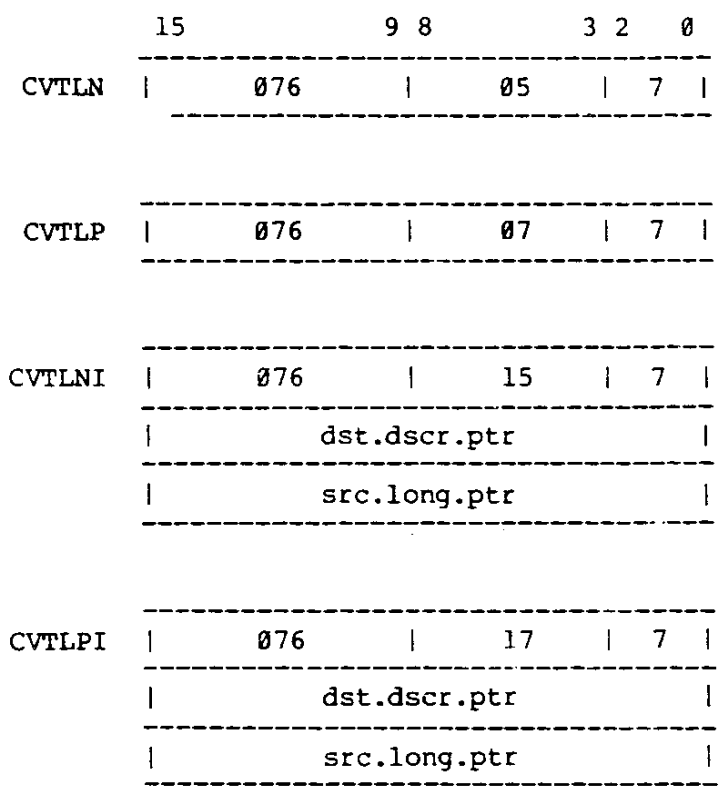
```
CMPNI / CMPPI           ; compare
.WORD   SRC1.DSCR.PTR   ; ptr to src1 descriptor
.WORD   SRC2.DSCR.PTR   ; ptr to src2 descriptor
BLT     NEGATIVE        ; negative destination
BEQ     EQUAL           ; zero destination
BGT     GREATER         ; positive destination
```

Notes:

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

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

Format:



Operation:

decimal string <- long integer

Condition Codes:

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

Suspendability:

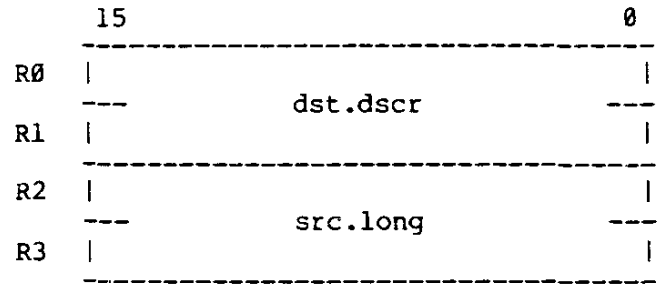
This instruction is potentially suspendable.

Description:

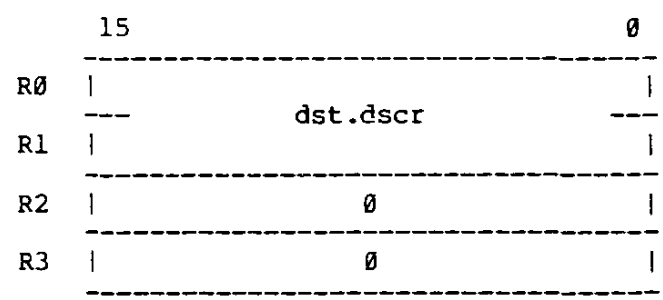
The source long integer is converted to a decimal string. The condition codes reflect the result stored in the destination decimal string, and whether all significant digits were stored.

Register Form - CVTLN and CVTLP

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



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



In-line Form - CVTLNI and CVTLPI

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

Formal Description:

TBS;

Examples:

1. Convert Long to Decimal - Register Form

```
MOV      DST.DSCR,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
```

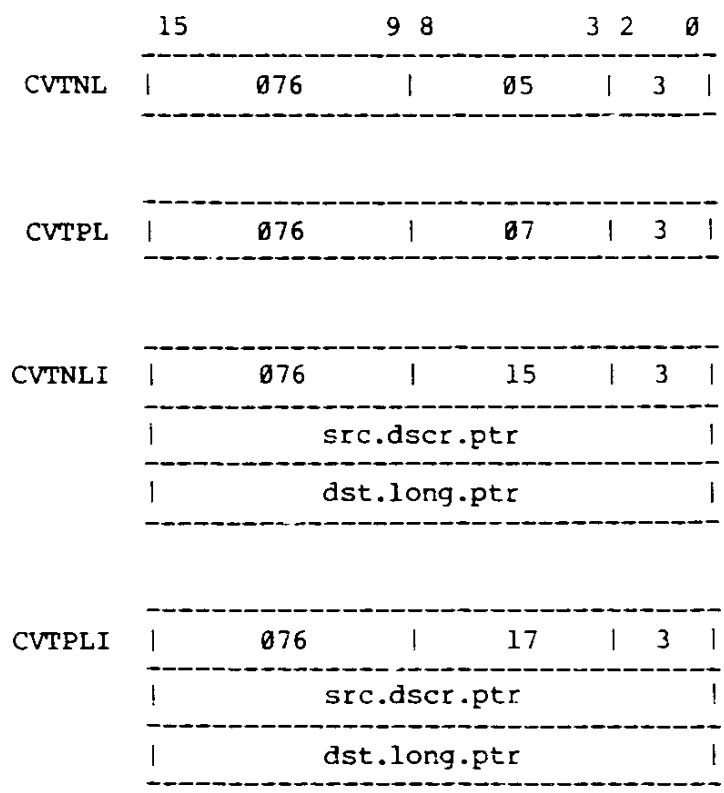
Notes:

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:



Operation:

long integer <- decimal string

Condition Codes:

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

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

Suspendability:

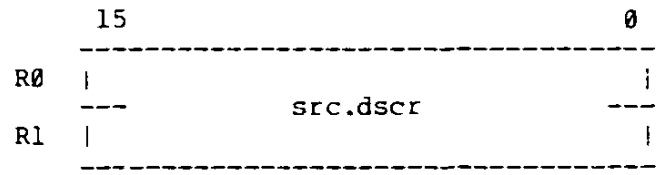
This instruction is potentially suspendable.

Description:

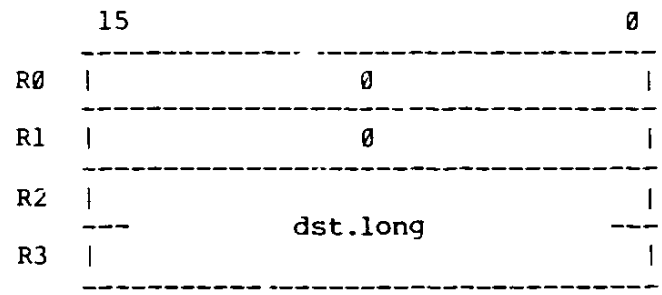
The source decimal string is converted to a long integer. The condition codes reflect the result of the operation, or whether significant digits were not converted.

Register Form - CVTNL and CVTPL

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



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



In-line Form - CVTNLI and CVTPLI

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

Formal Description:

TBS;

Examples:

1. Convert Decimal to Long - Register Form

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

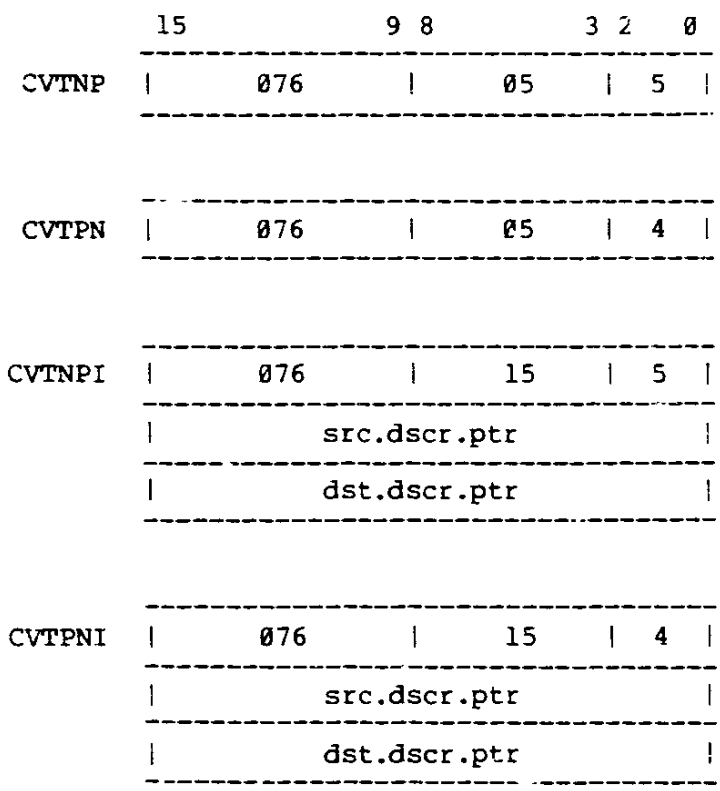
2. In-line forms use a long integer oriented with the low order portion in dst.long, and the sign and high order portion in dst.long+2.

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:



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.

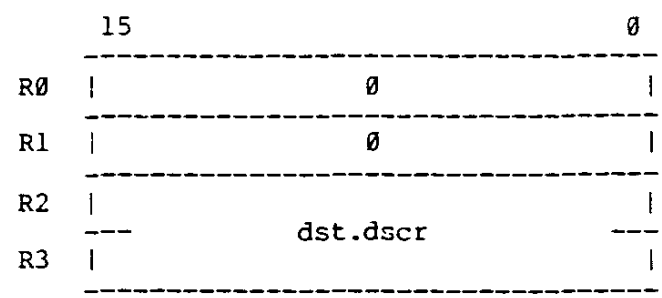
Description:

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:

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



In-line Form - CVTNPI and CVTPNI

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

Formal Description:

TBS;

Examples:

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

```
        MOV     SPC.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
```

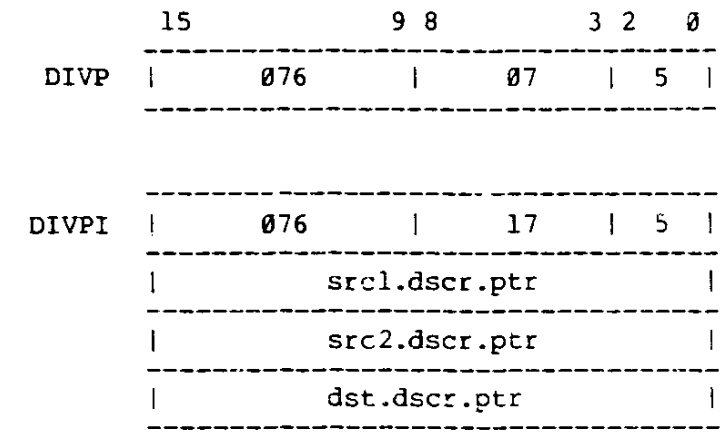
Notes:

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:



Operation:

dst <- src2 / src1

Condition Codes:

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

Suspendability:

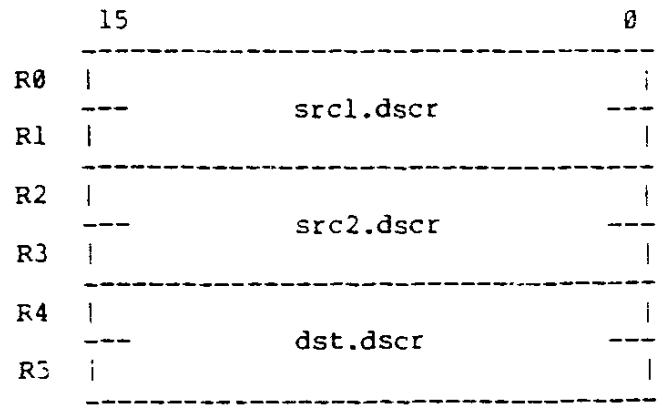
This instruction is potentially suspendable.

Description:

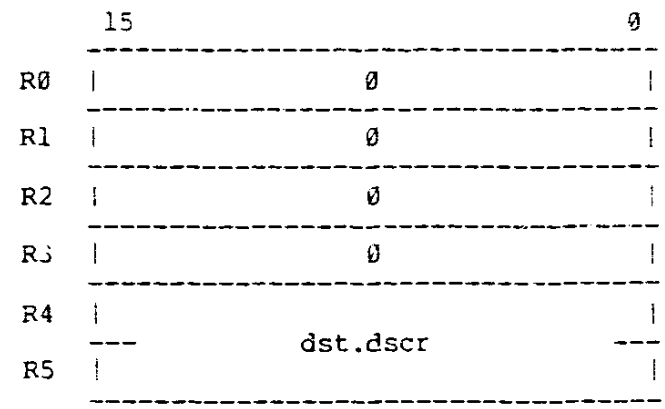
Src2 is divided by src1, and the quotient (fraction truncated) is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - DIVP

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



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



In-line Form - DIVPI

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

Formal Description:

TBS;

Examples:

1. Divide - Register Form

```
MOV     SRC1.DSCR,R0    ; 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.

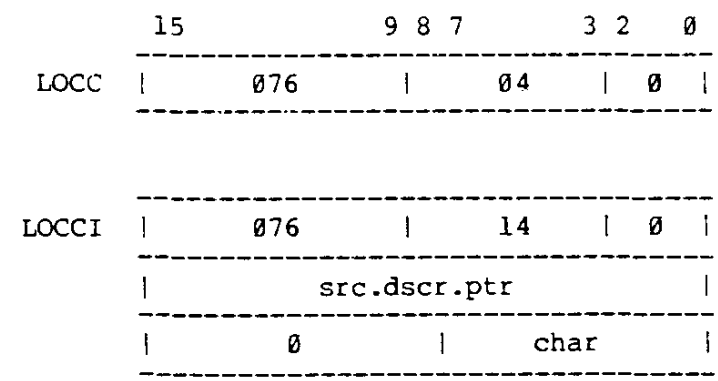
2. The results of the instruction are unpredictable if the source and destination strings overlap.

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:



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:

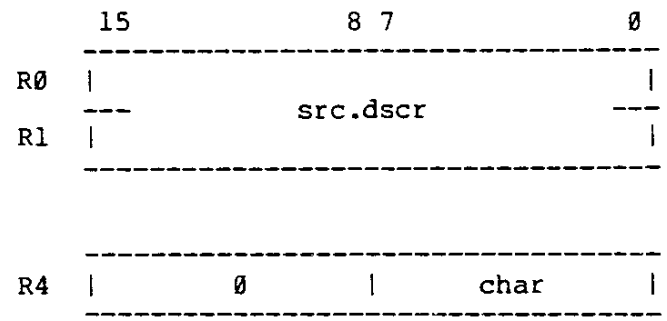
This instruction is potentially suspendable.

Description:

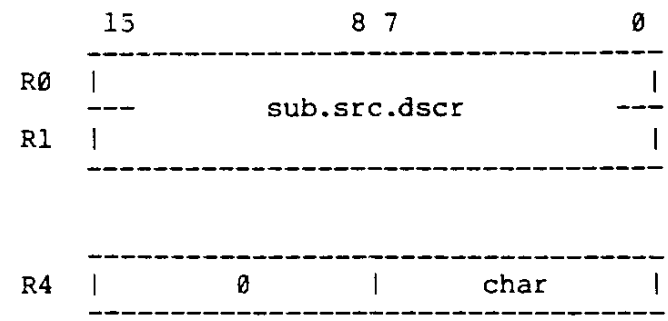
The source character string is searched from most significant to least significant character until the first occurrence of the search character. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with the located character. If the 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 RØ-R1, the search character is placed in R4<7:Ø>, and R4<15:8> must be zero:

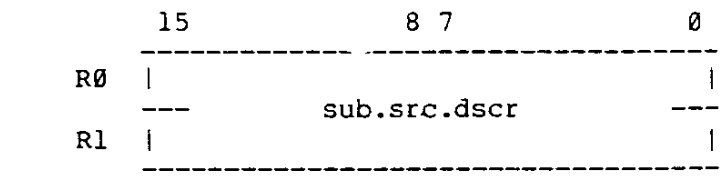


When the instruction is completed, RØ-R1 contain a character set descriptor which represents the sub-string of the source character string beginning with the located character:



In-line Form - LOCCI

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



Formal Description:

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

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

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

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

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

Examples:

1. Find the Beginning of a Comment - Register Form

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

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

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

Notes:

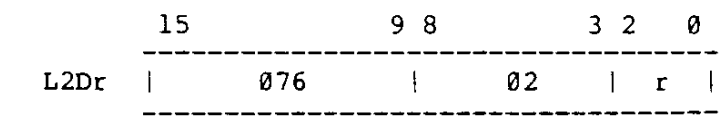
1. If the initial source character string descriptor is vacant, the instruction terminates with the condition codes indicating no match was found. The original source character string descriptor is returned in R0-R1.

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

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

5.10 L2Dr - Load 2 Descriptors

Format:



Operation:

Load word pairs into 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:

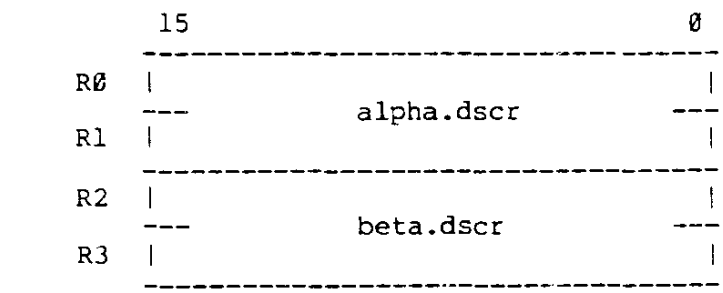
This instruction is non-suspendable.

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-Rl and the 'beta' descriptor is in R2-R3:



Formal Description:

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

Examples:

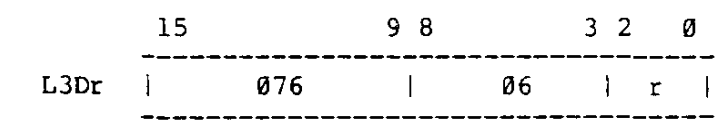
1. Decimal String Compare

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

Notes:

5.11 L3Dr - Load 3 Descriptors

Format:



Operation:

Load word pairs into R0-R1, 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:

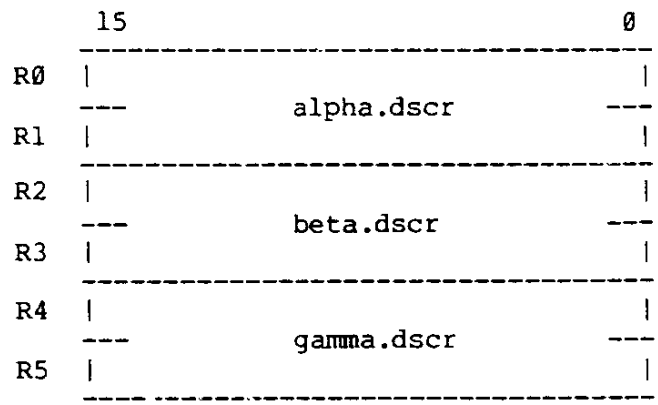
This instruction is non-suspendable.

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 R0-R1; a second descriptor 'beta' is loaded into R2-R3; a third descriptor 'gamma' is loaded into R4-R5. The address of the descriptors are determined by the addressing mode @(Rr)+ where r is the low order three bits of the opcode word. The address of the descriptor 'alpha' is derived by applying this addressing mode once; the address of the descriptor 'beta' is derived by applying this addressing mode a second time; the address of the descriptor 'gamma' is derived by applying this addressing mode a third time. The address mode auto-increments the indicated register by 2. The addressing mode computation is not affected by the descriptors which are loaded into the general registers. The words which contain the addresses of the descriptors are in consecutive words in memory; the descriptors themselves may be anywhere in memory. The condition codes are not affected.

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



Formal Description:

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

Examples:

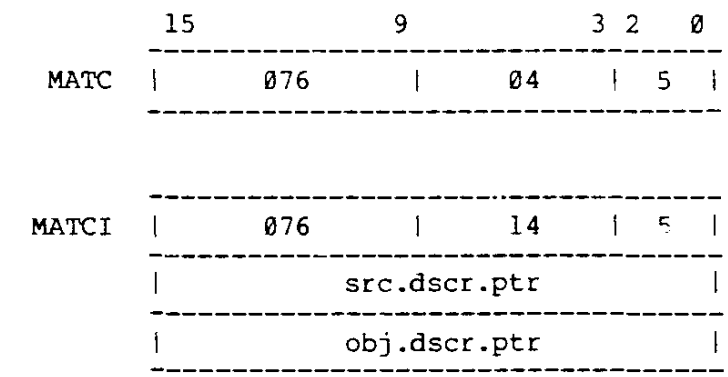
1. Three Address Add

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

Notes:

5.12 MATC / MATCI - Match Character

Format:



Operation:

Search source character string for object character string.

Condition Codes:

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

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

Suspendability:

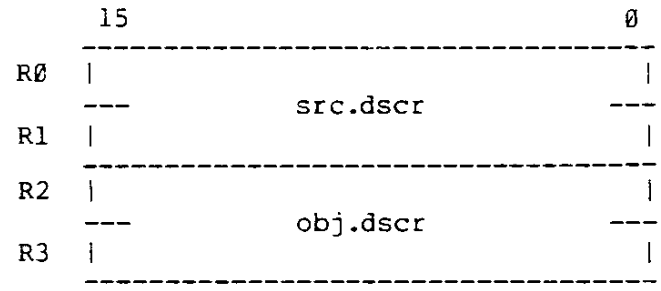
This instruction is potentially suspendable.

Description:

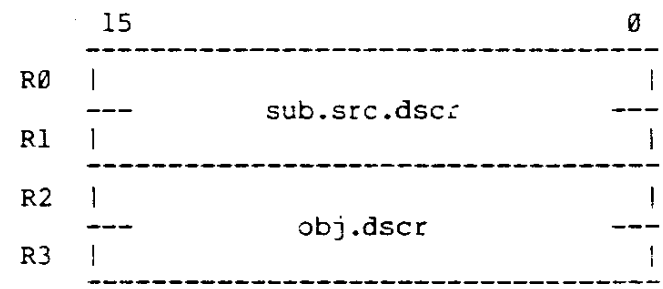
The source character string is searched from most significant to least significant character for the first occurrence of the entire object character string. A character string descriptor is returned in R0-R1 which represents the portion of the original source character string beginning with the most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

Register Form - MATC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in RØ-R1, and the object character string descriptor is placed in R2-R3:

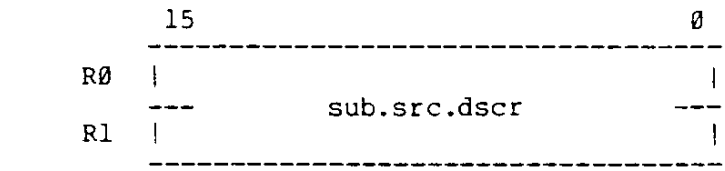


The instruction terminates with a character sub-string descriptor returned in RØ-R1 which represents the portion of the original source character string beginning with the most significant character to completely match the object character string.



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.



Formal Description:

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

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

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

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

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

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

Examples:

1. Find a Keyword - Register Form

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

2. Find a Keyword - In-line Form

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

Notes:

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

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

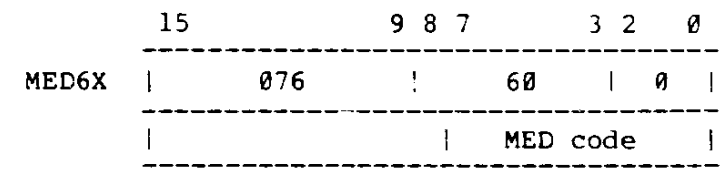
3. If the length of the object character string is greater than that of the source character string then no match is found; R0-R1 and the condition codes will be updated.

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

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

5.13 MED6X - PDP11/60 Maintenance, Examine, Deposit

Format:



Operation:

Access to internal processor registers.

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 reads or writes an internal processor register on the PDP11/60.

R0 is an implicit operand and either contains the source data which is to be written into an internal processor register or serves as the destination for a read-operation from an internal processor register. For MED codes 154 and 155, R2 and R3 also serve as implicit operands, as shown in the table below.

The explicit opcode specific operand which immediately follows the opcode in the instruction stream defines whether the operation is a "read" or a "write" and it specifies an internal processor address by which the internal processor register can be accessed. Bits <15:8> of this operand are ignored.

The condition codes are not affected.

The following table details this operation. In the table "xxx" indicates that the value of the high byte is a "don't care". The effects of executing unspecified operations are unpredictable.

Operand	Operation
xxx00n	read low half of A scratch pad Low, word n
xxx01n	read high half of A scratch pad Low, word n
xxx02n	read low half of A scratch pad High, word n
xxx03n	read high half of A scratch pad High, word n
xxx04n	read low half of B scratch pad Low, word n
xxx05n	read high half of B scratch pad Low, word n
xxx06n	read low half of B scratch pad High, word n
xxx07n	read high half of B scratch pad High, word n
xxx10n	read C scratch pad, word n
xxx11n	read C scratch pad, word 10(8)+n
xxx140	read Jam register
xxx141	read Service register
xxx142	read Physical (Unibus) Address register
xxx143	read Current Micro-Address register
xxx144	read Flag register
xxx145	NOP
xxx146	read Revision register
xxx147	read Count register
xxx152	read Diagnostic Control Store register 1
xxx153	read Diagnostic Control Store register 2
xxx154	Invalidate cache location corresponding to physical address in R3 and R2, where R3<1:0> contains bits <17:16> of the physical address and R2 contains bits <15:0> of the physical address
xxx155	read Cache Tag corresponding to the physical address in R3 and R2, where R3<1:0> contains bits <17:16> of the physical address and R2 contains bits <15:0> of the physical address
xxx20n	write low half of A scratch pad Low, word n
xxx21n	write high half of A scratch pad Low, word n
xxx22n	write low half of A scratch pad High, word n
xxx23n	write high half of A scratch pad High, word n
xxx24n	write low half of B scratch pad Low, word n
xxx25n	write high half of B scratch pad Low, word n
xxx26n	write low half of B scratch pad High, word n
xxx27n	write high half of B scratch pad High, word n
xxx30n	write C scratch pad, word n
xxx31n	write C scratch pad, word 10(8)+n
xxx344	write Flag register
xxx345	write D register
xxx346	write Shift register
xxx347	write Counter register
xxx350	write Next Micro-Address register
xxx351	write Residual Control register
xxx352	write Init register
xxx353	NOP

Formal Description:

TBS;

Examples:

1. Log abort-type error condition

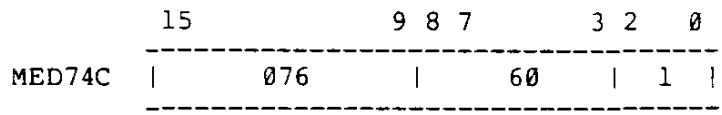
```
        MOV     #LOGBUF,R1
        MED     #100            ; on abort-type error
        MOV     R0,(R1)+        ; condition move
    internal
        MED     #101            ; machine state to
        MOV     R0,(R1)+        ; error logging buffer
          .
          .
          .
```

Notes:

1. This is a reserved instruction in User Mode.

5.14 MED74C - PDP11/74 CIS Maintenance Instruction

Format:



Operation:

CIS next micro-address <- R5<12:0>

Condition Codes:

The condition codes will be set by the PDP11/74 CIS processor.

Suspendability:

This instruction is potentially suspendable.

Description:

This is a maintenance and diagnostic instruction for the PDP11/74 CIS processor. It suspends operation of the PDP11/74 base machine, and initiates operation of the PDP11/74 CIS processor by loading its next micro-address register. The micro-address is in R5<12:0>; R5<15:13> is ignored. The effect of this instruction is dependent upon the micro-program which is executed.

Formal Description:

CIS.processor.next.micro.address.reg = R5<12:0>;

Examples:

1. Transfer control to CIS processor.

```
MOV     NONZERO,@#177770 ; set micro-break
MOV     MICRO.ADR,R5     ; CIS micro-address
MED74C                   ; transfer control
```

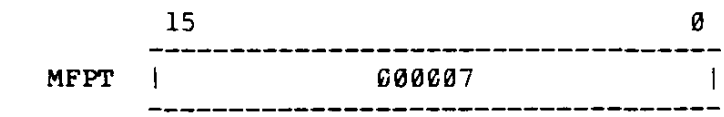
Notes:

1. This instruction is reserved if the high order 3 bits of the PDP11/74 micro-program break register are cleared (PB<15:13>). The micro-program break register is at physical address 17777770(8); it is cleared during processor power-up, manual activation of the front panel start switch, or successful execution of a RESET instruction.

2. Refer to maintenance documentation for the values which are obtained when reading the contents of the micro-program break register.

5.15 MFPT - Move From Processor Type

Format:



Operation:

```
R0<7:0>  <-- processor model code
R0<15:8> <-- processor subcode
```

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:

No source operands are used.

Upon execution, the MFPT instruction returns in the low byte of R0 a processor model code, as specified in the table below. The high byte of R0 will be loaded with a processor specified subcode.

The condition codes are not affected.

The previous contents of R0 are lost.

The codes to be returned in the low byte of R0 are as follows:

code (octal)	processor type
TBS	TBS

Formal Description:

```
R0<7:0> = processor.model.code;
R0<15:8> = processor.subcode;
```

Examples:

1. Get processor type-code

```
MOV     R0,-(SP)        ; save R0
MFPT                    ; get processor model
MOVB    R0,CPUTYP       ; store it
MOV     (SP)+,R0        ; restore R0
```

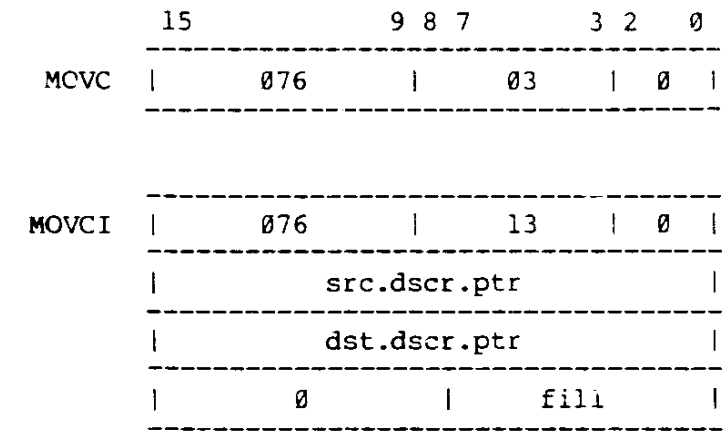
Notes:

1. On processors where this instruction is not implemented, a reserved instruction trap through vector 10(8) is taken.

2. The processor model codes are assigned by the PDP11 Architecture Group Manager. This standard will be updated to include the model codes for processors which have been publicly announced. Codes for processors under development may also have been assigned.

5.16 MOVC / MOVCI - Move Character

Format:



Operation:

dst <- src

Condition Codes:

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

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

Suspendability:

This instruction is potentially suspendable.

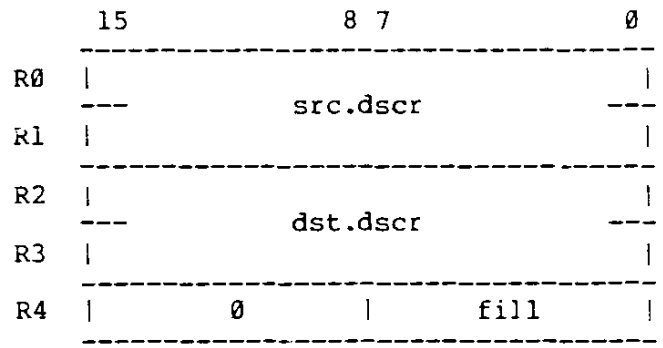
Description:

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

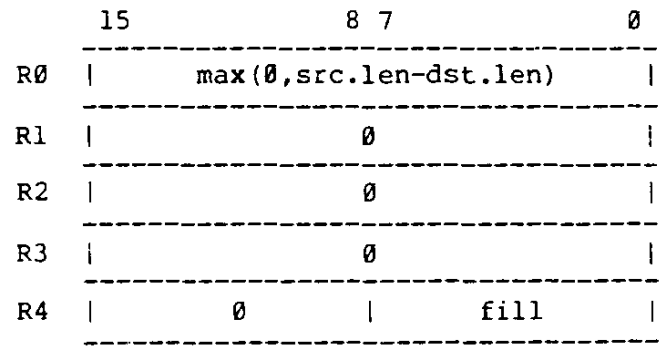
If the source string is longer than the destination string, the least significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This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

Register Form - MOVC

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



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



In-line Form - MOVCI

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

Formal Description:

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

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

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

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

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

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

Examples:

1. Moving Data - Register Form

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

2. Moving Data - In-line Form

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

3. Clearing Storage - Register Form

```
CLR     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.

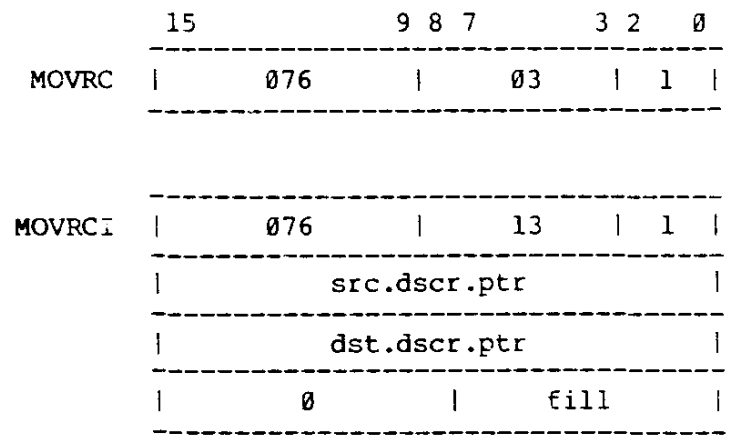
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. The condition codes will be updated. MOVC will update the general registers.

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:



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.

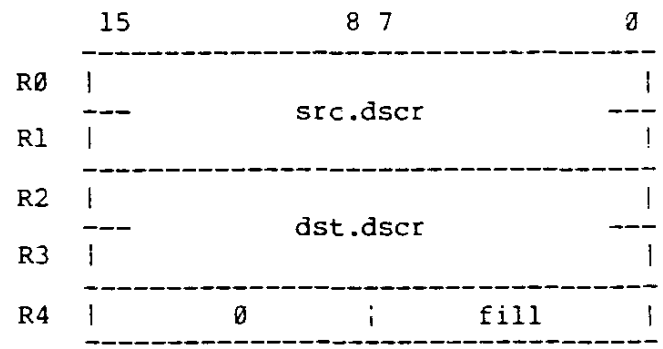
Description:

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.

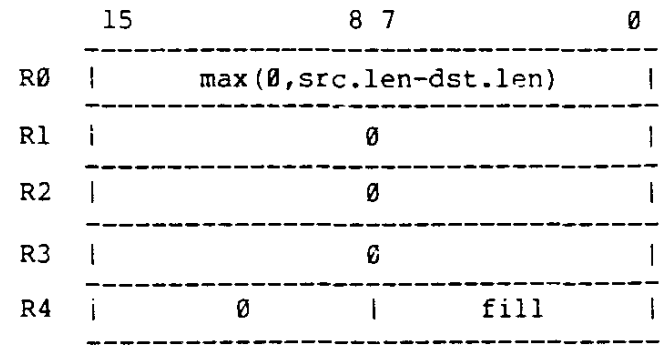
If the source string is longer than the destination string, the most significant characters of the source string are not moved. This is indicated by the Z and C bits cleared. If the source and destination strings are of equal length, all characters are moved with neither truncation nor filling. This 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:



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



In-line Form - MOVRCI

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

Formal Description:

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

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]<7:0>;     !     .
R7 = R7+2;             !     .

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

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

```
            end;
        end
    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
```

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.

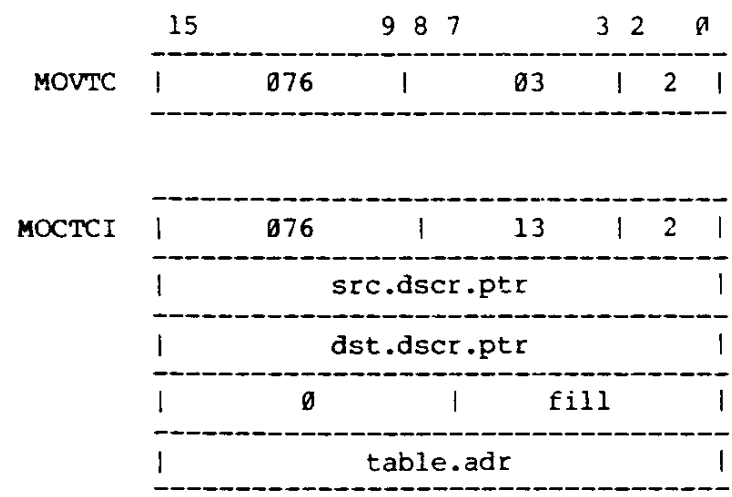
2. If the source string is vacant, the fill character will be propagated through the destination string. If the destination string is vacant, no characters will be moved. Condition codes will be updated. MOVRC will update the general registers.

3. MOVRC -- When the instruction terminates, 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.

5.18 MOVTC / MOVTCI - Move Translated Character

Format:



Operation:

dst <- translated src

Condition Codes:

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

N: set if result<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.

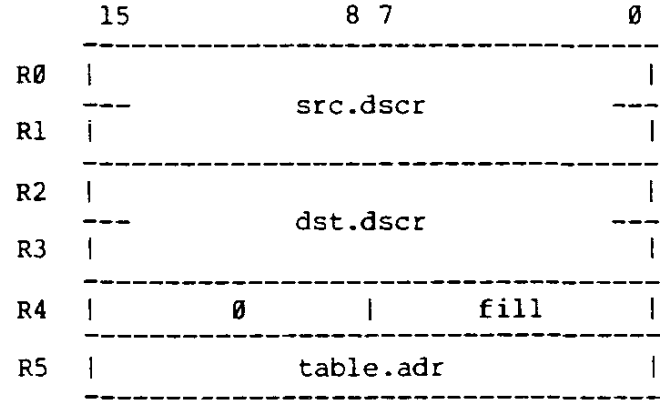
Description:

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.

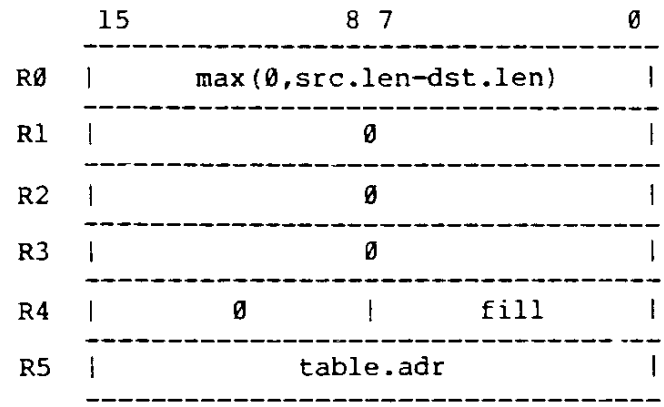
If the source string is shorter than the destination string, the untranslated fill character is used to complete the least significant part of the destination string. This is indicated by the C bit set. If the source string is longer than the destination string, the least significant characters of the 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 R0-R1, the destination character string descriptor is placed in R2-R3, the fill character is placed in R4<7:0>, R4<15:8> must be zero, and the translation table address is placed in R5:



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



In-line Form - MOVTCI

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

Formal Description:

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

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

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

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

if src.adr gequ dst.adr then
        begin               ! most to least significant
 characters
        while (src.len nequ 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;       !    .
```

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.

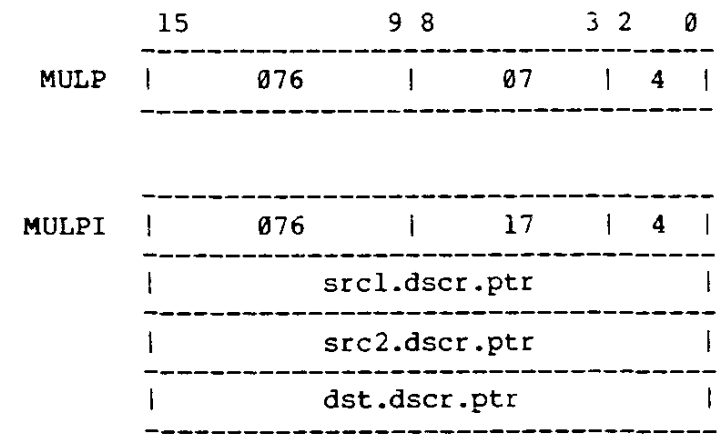
4. MOVTC -- When the instruction terminates, RØ is zero only if Z or C are set.

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:



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:

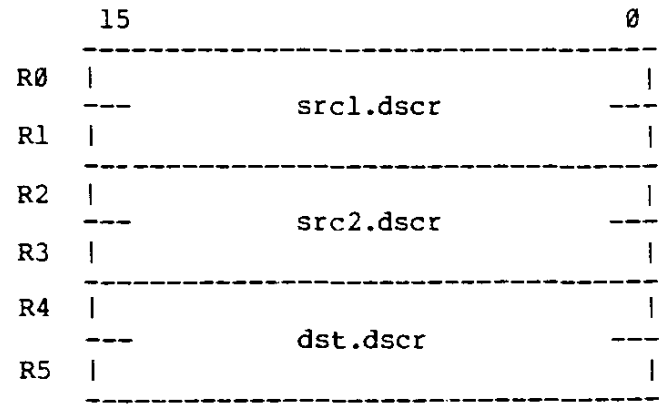
This instruction is potentially suspendable.

Description:

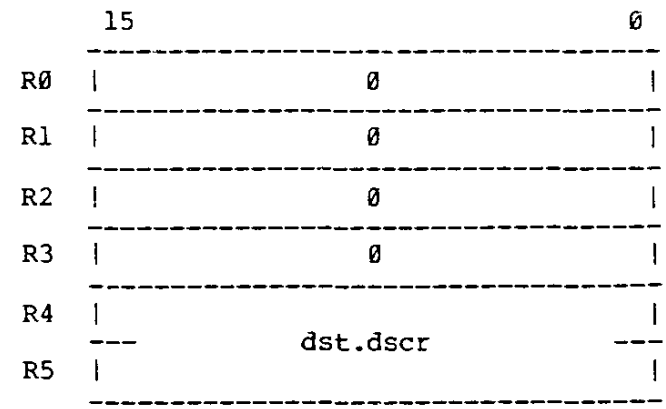
Src1 and src2 are multiplied, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

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:



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



In-line Form - MULPI

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

Formal Description:

TBS;

Examples:

1. Multiply - Register Form

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

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

2. Multiply - In-line Form

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

Notes:

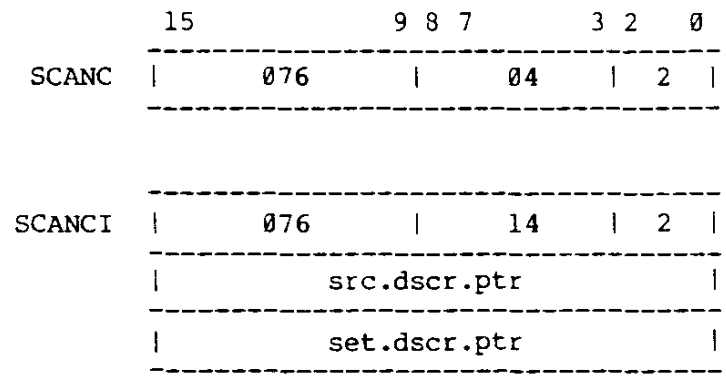
1. The operation of these instructions is unaffected by any overlap of the source strings provided that each source string is a valid representation of the specified data type.

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

3. No numeric string multiply instruction is provided.

5.20 SCANC / SCANCI - Scan Character

Format:



Operation:

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

Condition Codes:

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

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

Suspendability:

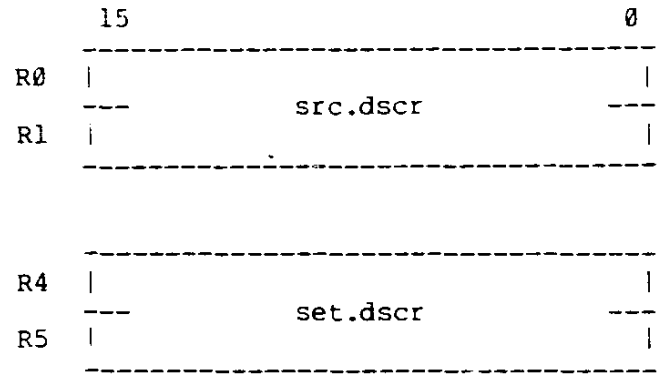
This instruction is potentially suspendable.

Description:

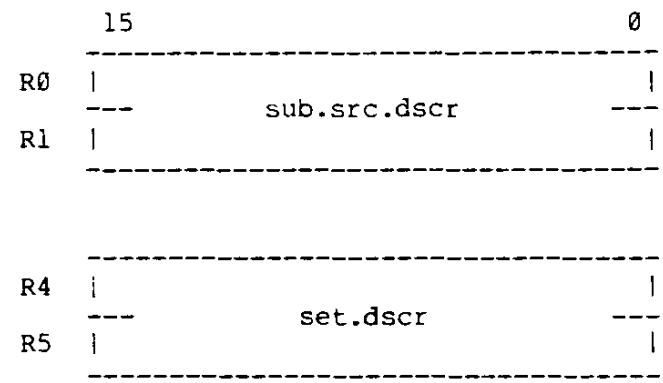
The source character string is searched from most significant to least significant character until the first occurrence of a character which is a member of the character set. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning with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

Register Form - SCANC

When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5:

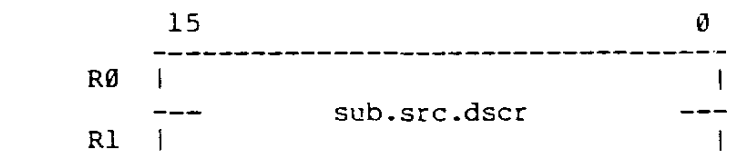


When the instruction is completed, R0-R1 contain a character string descriptor which represents the sub-string of the source character string beginning with the character which is a member of the character set:



In-line Form - SCANCI

The words which follow the opcode word in the instruction stream are a word address pointer to a two word character string source descriptor, and a word address pointer to a two word character set descriptor. When the instruction is completed, 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:



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   0                   ; ASCII 000
        .BYTE   0                   ; ASCII 001
        .BYTE   0                   ; ASCII 002
                                        .
                                        .
                                        .
        .BYTE   1                   ; ASCII 060 = '0'
        .BYTE   1                   ; ASCII 061 = '1'
        .BYTE   1                   ; ASCII 062 = '2'
        .BYTE   1                   ; ASCII 063 = '3'
        .BYTE   1                   ; ASCII 064 = '4'
        .BYTE   1                   ; ASCII 065 = '5'
        .BYTE   1                   ; ASCII 066 = '6'
        .BYTE   1                   ; ASCII 067 = '7'
        .BYTE   1                   ; ASCII 070 = '8'
        .BYTE   1                   ; ASCII 071 = '9'
        .BYTE   0                   ; ASCII 072
        .BYTE   0                   ; ASCII 073
                                        .
                                        .
                                        .
        .BYTE   0                   ; 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 R0-R1.

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

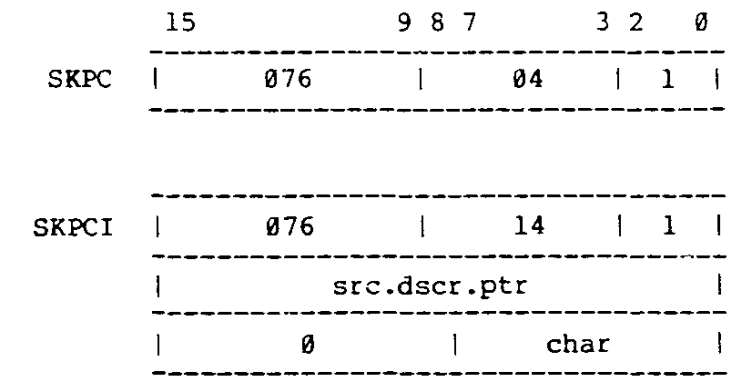
3. A test for success is BNE; a test for failure is BEQ.

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:



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 R0.

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

Suspendability:

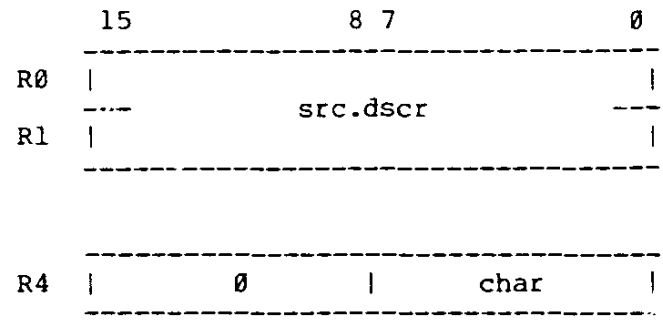
This instruction is potentially suspendable.

Description:

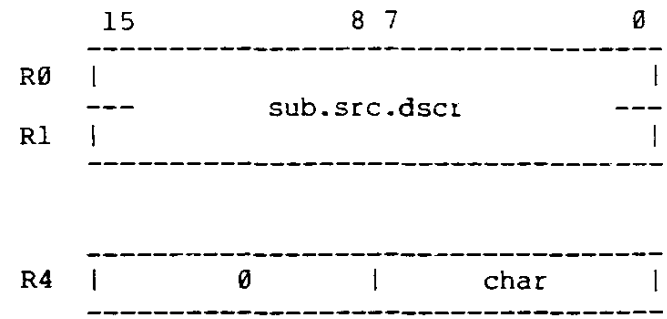
The source character string is searched from most significant to least significant character until the first occurrence of a character which is not the search character. A character string descriptor is returned in R0-R1 which represents the portion of the source character string beginning which the 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 R0.

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 R0-R1, the search character is placed in R4<7:0>, and R4<15:8> must be zero:

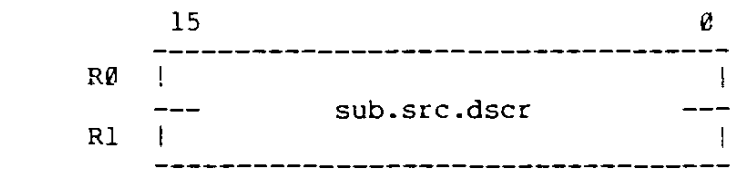


When the instruction is completed, R0-R1 contain a character string descriptor which represents the sub-string of the source character string beginning with the most significant character which was not equal to the search character:



In-line Form - SKPCI

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



Formal Description:

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

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

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

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

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

Examples:

1. Skip Leading Spaces -- Register Form

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

2. Skip Leading Spaces - In-line Form

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

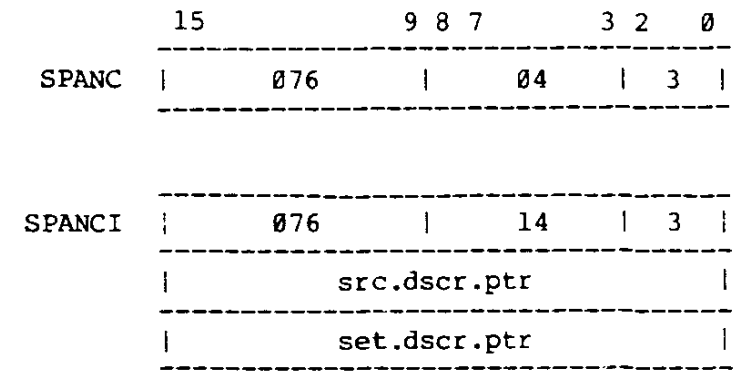
Notes:

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

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

5.22 SPANC / SPANCI - Span Character

Format:



Operation:

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

Condition Codes:

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

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

Suspendability:

This instruction is potentially suspendable.

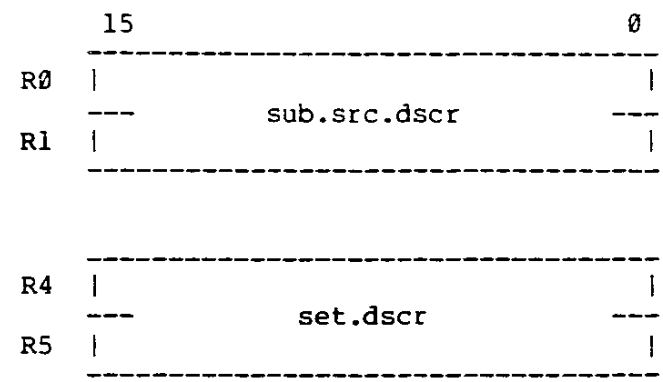
Description:

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

Register Form - SPANC

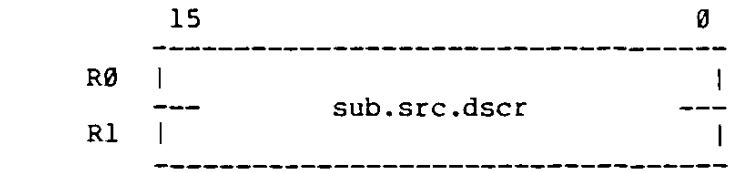
When the instruction starts, the operands must have been placed in the general registers. The source character string descriptor is placed in R0-R1, and the character set descriptor is placed in R4-R5:

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:



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:



Formal Description:

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

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

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

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

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

Examples:

1. Pass Tabs and Blanks - Register Form

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

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

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

2. Pass Tabs and Blanks - In-line Form

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

Notes:

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

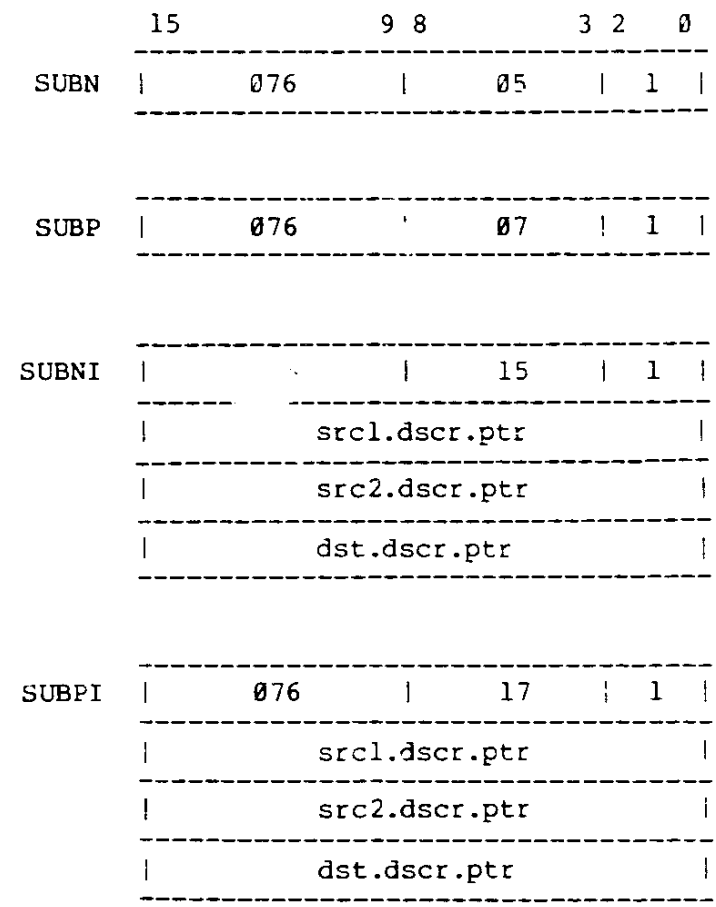
2. The source character string and character set table may overlap in any way.

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

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

5.23 SUBN / SUBP / SUBNI / SUBPI - Subtract Decimal

Format:



Operation:

dst <- src2 - src1

Condition Codes:

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

Suspendability:

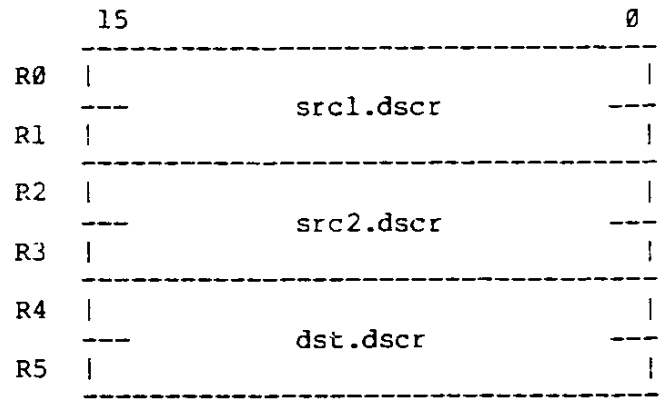
This instruction is potentially suspendable.

Description:

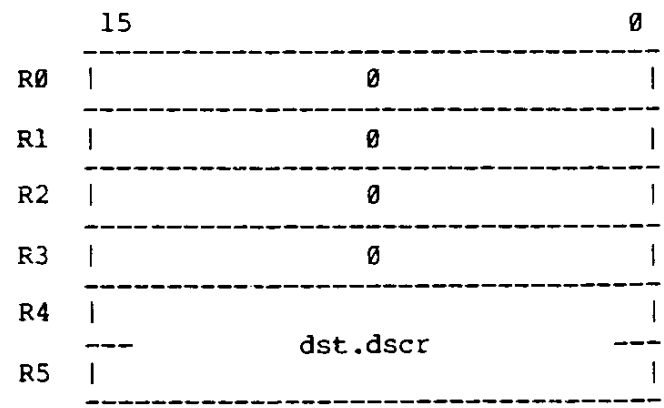
Src1 is subtracted from src2, and the result is stored in the destination string. The condition codes reflect the value stored in the destination string, and whether all significant digits were stored.

Register Form - SUBN and SUBP

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



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



In-line Form - SUBNI and SUBPI

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

Formal Description:

TBS;

Examples:

1. Three address subtract - Register Form

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

2. Three address subtract - In-line Form

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

3. Two address subtract - Register Form

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

4. Two address subtract - In-Line Form

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

Notes:

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

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

CHAPTER 6

Reinterpretation of Traditional PDP11 Instructions

6.1 MULTIPROCESSING MEMORY LOCK

The traditional ASRB instruction is now expected to be used for setting semaphores in main memory which can be used by the processor to "lock" access rights to designated system resources. In order to serve this function, the ASRB instruction must be so implemented as to ensure that the memory location specified as its operand be inaccessible to any other processor on the system during the interval in which the ASRB is executing.

If the machine is cached, the "copy" of the operand which the ASRB accesses and modifies must be the same "copy" as is directly visible to other processors on the system. This may mean that a cache-miss be forced.

The PDP11/70 and PDP11 compatibility mode on VAX11 machines do not implement the ASRB lock facility.

CHAPTER 7

Waivers

7.1 PDP11/60 Lacking MFPT

Date

requested -- 11-Nov-76
approved --

Requestor

Bob Magers (76BM373-1543)

Relevant Sections of this Standard

Sections 1.4.3 and 4.1.

Description of the Waiver

The PDP11/60 (KD11-K) will not implement the MFPT instruction.

Reasons Justifying the Waiver

The PDP11/60 design and development anteceded the final formulation, review, and approval of this Standard. Retrofit implementation would imact project schedule. Current software does not yet require or support the MFPT instruction. A retrofit ECO is possible in the future.

7.2 LSI-11 Commercial Instruction Set

Date

Requested -- 1-May-77, 14-Jul-78
Approved --

Requestor

Ralph Platz

Relevant Sections of this Standard

Sections 4.2 and 4.3.

Description of the Waiver

The LSI-11 uses a non-zero value in the high byte of R4 to indicate instruction suspension; PS<8> is not implemented.

The LSI-11 does not implement all instructions in the closed groups for character string moves, character string searches, and numeric strings. Only the register forms of instructions are provided. No packed string or load descriptor instructions are provided. All unimplemented opcodes and features trap through vector 10(8); the entire user visible state is unchanged (except for R6 and the PC and PS which are pushed on the stack).

In the Character String Move Group, MOVC and MOVRC are implemented according to this specification. MOVTC is not implemented.

In the Character String Search Group, LOCC, SKPC, SCANC, SPANC and CMPC are implemented according to this specification. MATCHC is not implemented.

In the Numeric String Group, only the signed zoned decimal string data type is supported. Zero length source operands cause instructions to trap through vecotr 10(8). CVTNL produces unpredictable results in R2, R3, N and Z if V is set. Other than the stated considerations resulting from the limitations in data type, data length and overflow, the ADD, SUBN, CMPN and CVTNL are implemented according to this specification. CVTPN, CVTNP, ASHN and CVTLN are not implemented.

Reasons Justifying the Waiver

The LSI-11 Commercial Instruction Set implementation was in progress before this specification had been finalized. Architectural restrictions reflect the constraints of limited micro-code expandability, product cost and performance requirements. Zero length zoned source strings trap because of an OOD decision in force at the time the micro-code was committed.

The use of PS<8> to indicate instruction suspension was adopted after the LSI-11 implementation was completed.

APPENDIX A

Extended-Instruction Opcode Assignments

Opcode	Mnemonic	Instruction	N	Z	V	C
Included in Basic Instruction Set						
000007	MFPT	move from processor type	-	-	-	-
Commercial Load 2 Descriptors						
076020	L2D0	load 2 descriptors @(R0)+	-	-	-	-
076021	L2D1	load 2 descriptors @(R1)+	-	-	-	-
076022	L2D2	load 2 descriptors @(R2)+	-	-	-	-
076023	L2D3	load 2 descriptors @(R3)+	-	-	-	-
076024	L2D4	load 2 descriptors @(R4)+	-	-	-	-
076025	L2D5	load 2 descriptors @(R5)+	-	-	-	-
076026	L2D6	load 2 descriptors @(R6)+	-	-	-	-
076027	L2D7	load 2 descriptors @(R7)+	-	-	-	-
Character String Move						
076030	MOVC	move character	*	*	*	*
076031	MOVRC	move reverse character	*	*	*	*
076032	MOVTC	move translated (Character)	*	*	*	*
076033		reserved				
076034		reserved				
076035		reserved				
076036		reserved				
076037		reserved				
Character String Search						
076040	LOCC	locate character	*	*	0	0
076041	SKPC	skip character	*	*	0	0
076042	SCANC	scan character	*	*	0	0
076043	SPANC	span character	*	*	0	0
076044	CMPC	compare character	*	*	*	*
076045	MATC	match character	*	*	0	0
076046		reserved				
076047		reserved				
Numeric String						
076050	ADDN	add numeric	*	*	*	0
076051	SUBN	subtract numeric	*	*	*	0
076052	CMPN	compare numeric	*	*	0	0
076053	CVTNL	convert numeric to long	*	*	*	*
076054	CVTPN	convert packed to numeric	*	*	*	0
076055	CVTNP	convert numeric to packed	*	*	*	0
076056	ASHN	arithmetic shift numeric	*	*	*	0
076057	CVTLN	convert long to numeric	*	*	*	0

Commercial Load 3 Descriptors

076060	L3D0	load 3 descriptors @(R0)+	- - - -
076061	L3D1	load 3 descriptors @(R1)+	- - - -
076062	L3D2	load 3 descriptors @(R2)+	- - - -
076063	L3D3	load 3 descriptors @(R3)+	- - - -
076064	L3D4	load 3 descriptors @(R4)+	- - - -
076065	L3D5	load 3 descriptors @(R5)+	- - - -
076066	L3D6	load 3 descriptors @(R6)+	- - - -
076067	L3D7	load 3 descriptors @(R7)+	- - - -

Packed String

076070	ADDP	add packed	* * * 0
076071	SUBP	subtract packed	* * * 0
076072	CMPP	compare packed	* * 0 0
076073	CVTPL	convert packed to long	* * * *
076074	MULP	multiply packed	* * * 0
076075	DIVP	divide packed	* * * *
076076	ASHP	arithmetic shift packed	* * * 0
076077	CVTLP	convert long to packed	* * * 0

Character String Move (in-line)

076130	MOVCI	move character	* * * *
076131	MOVRCI	move reverse character	* * * *
076132	MOVTCI	move translated character	* * * *
076133		reserved	
076134		reserved	
076135		reserved	
076136		reserved	
076137		reserved	

Character String Search (in-line)

076140	LOCCI	locate character	* * 0 0
076141	SKPCI	skip character	* * 0 0
076142	SCANCI	scan character	* * 0 0
076143	SPANCI	span character	* * 0 0
076144	CMPCI	compare character	* * * *
076145	MATCI	match character	* * 0 0
076146		reserved	
076147		reserved	

Numeric String (in-line)

076150	ADDNI	add numeric	* * * 0
076151	SUBNI	subtract numeric	* * * 0
076152	CMPNI	compare numeric	* * 0 0
076153	CVTNLI	convert numeric to long	* * * *
076154	CVTPNI	convert packed to numeric	* * * 0
076155	CVTNPI	convert numeric to packed	* * * 0

076156	ASHNI	arithmetic shift numeric	* * * 0
076157	CVTNLI	convert long to numeric	* * * 0

Packed String (in-line)

076170	ADDPI	add packed	* * * 0
076171	SUBPI	subtract packed	* * * 0
076172	CMPPI	compare packed	* * 0 0
076173	CVTPLI	convert packed to long	* * * *
076174	MULPI	multiply packed	* * * 0
076175	DIVPI	divide packed	* * * *
076176	ASHPI	arithmetic shift packed	* * * 0
076177	CVTLPI	convert long to packed	* * * 0

Processor-Specific #0

076600	MED6X	PDP11/60 Maintainance	- - - -
076601	MED74C	PDP11/74 CIS Maintainance	* * * *
076602		reserved	
076603		reserved	
076604		reserved	
076605		reserved	
076606		reserved	
076607		reserved	

* conditionally set/cleared
\- not affected
0 cleared
1 set

APPENDIX B

PDP11 Opcode Space

Legend:

Note: Upper-case characters represent a full 3-bit octal digit; lower-case characters represent 1 or 2 bits.

Ss general source operand specifier (mode,register)
bits: <11:6>,<5:0>

Dd general destination operand specifier (mode,register)
bits: <5:0>

R register
bits: <8:6>,<5:3>,<2:0>

P field for SPL and microcode escape
bits: <2:0>

Nn count for SOB
bits: <5:0>

cC condition code states
bits: <3:0>

xXX branch offset
bits: <7:0>

iII immediate data in Emt and Trap instructions
bits: <7:0>

Fs floating-point source operand specifier (mode,register)
bits: <5:0>

Fd floating-point destination operand specifier (mode,register)
bits: <5:0>

a floating-point accumulator specifier
bits: <7:6>

space	opcode	mnemonic
	000000	HALT
	000001	WAIT
	000002	RTI
8.	000003	BPT
	000004	IOT
	000005	RESET
	000006	RTT
	000007	MFPT
	000007	----------
57.		reserved instruction space
	000077	----------
	0001DD	JMP
72.	00020R	RTS
	00021R	maintenance (LSI-11)
16.	00022P	escape to microcode (LSI-11)
8.	00023P	SPL
	000240	NOP
	0002(4+c)C	clear condition codes
	241	CLC
	242	CLV
	244	CLZ
	250	CLN
	257	CCC
32.	000260	(NOP)
	0002(6+c)C	set condition codes
	261	SEC
	262	SEV
	264	SEZ
	270	SEN
	277	SCC
64.	0003DD	SWAB
	000(4+x)XX	BR
	001(o+x)XX	BNE
	001(4+x)XX	BEQ
1792.	002(0+x)XX	BGE
	002(4+x)XX	BLT
	003(0+x)XX	BGT
	003(4+x)XX	BLE
512.	004RDD	JSR
	0050DD	CLR
	0051DD	COM

```
    |       0052DD       INC
    |       0053DD       DEC
    |       0054DD       NEG
  768.      0055DD       ADC
    |       0056DD       SBC
    |       0057DD       TST
    |       0060DD       ROR
    |       0061DD       ROL
    |       0062DD       ASR
    |       0063DD       ASL
-------
    |       0064DD       MARK
    |       0065DD       MFPI
  256.      0066DD       MTPI
    |       0067DD       SXT
-------
    |       007000  ----------------------------------
  512.                    reserved instruction space    |
    |       007777  ------------------------------- --
-------
    |       01SSDD       MOV
    |       02SSDD       CMP
    |       03SSDD       BIT
24576.      04SSDD       BIC
    |       05SSDD       BIS
    |       06SSDD       ADD
-------
    |       070RSS       MUL ---------
    |       071RSS       DIV          |
 2560.      072RSS       ASH      EIS |
    |       073RSS       ASHC---------
    |       074RDD       XOR
-------
    |       07500R       FADD---------
    |       07501R       FSUB         |
   32.      07502R       FMUL     FIS |
    |       07503R       FDIV---------
-------
    |       075040
  480.              maintenance (LSI-11)
    |       075777
-------
    |       076000 ------------------------------------------------
  512.           E X T E N D E D - I N S T R U C T I O N  S P A C E |
    |       076777 ----------------------------------  -----------
-------
  512.      077RNN       SOB
-------
    |       100(0+x)XX   BPL
    |       100(4+x)XX   BMI
    |       101(0+x)XX   BHI
    |       101(4+x)XX   BLOS
```

```
 2560.      102(0+x)XX  BVC
    |       102(4+x)XX  BVS
    |       103(0+x)XX  BCC,BHIS
    |       103(4+x)XX  BCS,BLO
    |       104(0+i)II  EMT
    |       104(4+i)II  TRAP
-------
    |       1050DD      CLRB
    |       1051DD      COMB
    |       1052DD      INCB
    |       1053DD      DECB
    |       1054DD      NEGB
    |       1055DD      ADCB
  768.      1056DD      SBCB
    |       1057DD      TSTB
    |       1060DD      RORB
    |       1061DD      ROLB
    |       1062DD      ASRB
    |       1063DD      ASLB
-------
    |       1064SS      MTPS
    |       1065SS      MFPD
  256.      1066DD      MTPD
    |       1067DD      MFPS
-------
    |       107000  ----------------------------------
  512.              |   reserved instruction space    |
    |       107777  ----------------------------------
-------
    |       11SSDD      MOVB
    |       12SSDD      CMPB
    |       13SSDD      BITB
24576.      14SSDD      BICB
    |       15SSDD      BISB
    |       16SSDD      SUB
-------
    |       170000      CFCC
    3.      170001      SETF
    |       170002      SETI
-------
    |       170003      ;maintenance
    3.      170004      maintenance
    |       170005      maintenance
-------             ------------------------------------------
    1.      170006      reserved floating point instruction  |
-------             ------------------------------------------
    1.      170007      maintenance
-------             ------------------------------------------
    1.      170010      reserved floating point instruction  |
-------             ------------------------------------------
```

```
    |       170011      SETD
    2.      170012      SETL
-------
    |       170013  ------------------------------------------
   53.                    reserved floating point instructions  |
    |       170077  ------------------------------------------
-------
    |       1701SS      LDFPS
    |       1702DD      STFPS
    |       1703FD      STST
    |       1704FD      CLRF,CLRD
    |       1705FD      TSTF,TSTD
    |       1706FD      ABSF,ABSD
    |       1707FD      NEGF,NEGD
    |       171(0+a)FS  MULF,MULD
    |       171(4+a)FS  MODF,MODD
 4032.      172(0+a)FS  ADDF,ADDD
    |       172(4+a)FS  LDF,LDD
    |       173(0+a)FS  SUBF,SUBD
    |       173(4+a)FS  CMPF,CMPD
    |       174(0+a)FD  STF,STD
    |       174(4+a)FS  DIVF,DIVD
    |       175(0+a)FD  STEXP
    |       175(4+a)FD  STCFI,STCFL,STCDI,STCDL
    |       176(0+a)FD  STCFD,STCDF
    |       176(4+a)FS  LDEXP
    |       177(0+a)FS  LDCIF,LDCID,LDCLF,LDCLD
    |       177(4+a)FS  LDCDF,LDCFD
-------
```

APPENDIX C

Formal Description of Machine State

```
! General Comments:
!
!    Details of this notation can be found in the
!    "ISPS Reference Manual".
!
!    All statements are followed by an implied NEXT.

! The following relational tests are used:
!
!    Two's Complement Comparisions
!
!        lss     less than
!        leq     less than or equal
!        eql     equal
!        neq     not equal
!        gtr     greater than
!        geq     greater than or equal
!
!    Unsigned comparisons
!
!        lssu    less than
!        lequ    less than or equal
!        eqlu    equal
!        nequ    not equal
!        gtru    greater than
!        gequ    greater or equal

! The max function returns the greatest of its arguments
! based on a two's complement comparison.

** Programmer.Visible.State **

     M[0:64K]<7:0>,   ! memory
     R0<15:0>,        ! general registers
     R1<15:0>,
     R2<15:0>,
     R3<15:0>,
     R4<15:0>,
     R5<15:0>,
     R6<15:0>,
     R7<15:0>,
     PS<15:0>,    ! processor status
         N<> := PS<3>,        ! condition codes
         Z<> := PS<2>,
         V<> := PS<1>,
         C<> := PS<0>,

** Temporary.State **
```

```
src.len<15:0>,
src.adr<15:0>,
obj.len<15:0>,
obj.adr<15:0>,
src1.len<15:0>,
src1.adr<15:0>,
src2.len<15:0>,
src2.adr<15:0>,
dst.len<15:0>,
dst.adr<15:0>,
part.len<15:0>,
part.adr<15:0>,
opr.1<15:0>,
opr.2<15:0>,
opr.3<15:0>,
opr.4<15:0>,
tmp.len<15:0>,
fill<7:0>,
char<7:0>,
mask<7:0>,
table.adr<15:0>,
temp<15:0>,
btmp<7:0>,
btmp1<7:0>,
btmp2<7:0>,
carry<>,
found<>,
alpha.adr<15:0>,
beta.adr<15:0>,
gamma.adr<15:0>
```