DECsystem-10/20 Hardware Manual

Stanford Artificial Intelligence Laboratory
Operating Note 75A

January 1977

DECsystem-10/20 Hardware Manual

by

Digital Equipment Corporation
Staff of the Artificial Intelligence Laboratory
Staff of the LOTS Computer Facility

Abstract

The Hardware Reference Manual explains the PDP-10 instruction set as it exists in the KL10 processor. Appendices explain the differences between the various processors in the PDP-10 family.

This manual is also published as part of LOTS Operating Note 2. This manual supersedes SAILON-26 and SAILON-71.

This manual was supported, in part, by the Advanced Research Projects Agency of the Department of Defense under Contract No. MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, the Advanced Research Projects Agency, the U.S. Government, or, for that matter, anyone else.

DEC-10-XSRMA-A-D
DEC-10-XSRMA-A-DN1

decsystem10

HARDWARE REFERENCE MANUAL

Direct comments concerning this manual to Software Quality Control,
Maynard, Massachusetts.

digital equipment corporation · maynard. massachusetts

Instruction times, operating speeds and the like are included here for reference only; they are not to be taken as specifications.

Copyright © 1968, 1969, 1971, 1974, 1976
by Digital Equipment Corporation

Changes are indicated by a triangle (▲) in the outside margin.

Fourth edition, March 1976

Manufactured in the United States of America

Preface

This manual explains the machine language programming and operation of the DECsystem-10, for both instructional and reference purposes. Basically the manual defines in detail how the central processor and the peripherals function, exactly what their instructions do, how they handle data, what their control and status information means, and what programming techniques and procedures must be employed to utilize them effectively. The programming is given in machine language, in that it uses only the basic instruction and device mnemonics and symbolic addressing defined by the assembler. The treatment relies on neither any other Digital software nor any of the more sophisticated features of the assembler; moreover the manual is completely self-contained – no prior knowledge of the assembler is required.

The text of the manual is devoted almost entirely to functional description and programming. Chapter 1 discusses the general characteristics of the system, defines the formats of the words used for numbers and instructions, and also explains the conventions needed to program the system and understand the examples given in the text. Chapter 2 covers all phases of the central processor, including the general principles of in-out programming and handling the interrupt system. The remaining chapters are devoted to the various categories of peripheral equipment. Chapters 3 and 4 cover the simple character-oriented devices that use form paper, paper tape and cards. Chapter 5 treats the data interfaces that are employed in the tape, disk and data communication systems covered in the three chapters following. Finally Chapter 9 describes the various terminals that can be used either at the console or in communication systems; this chapter includes both programming and operating information.

The first three appendices contain the basic reference tables for the programmer – word formats, instruction and device mnemonics, IO codes, IO bit assignments showing conditions and status, and a shorthand presentation of instruction actions in symbolic form. The next two appendices provide additional programming information of less general use: Appendix D gives the instruction times and Appendix E documents the differences among the several central processor models. The final three appendices provide a complete guide to the operation of the central processors, memories and peripheral devices (except terminals). This treatment is entirely in hardware terms, describing all lights and switches, how to load the devices, and so forth, but not how to run the system in terms of interacting with any Digital software – that information is given in the DECsystem-10 Operator's Guide.

Contents

Note: in the present publication chapters 3 through 8 have been omitted. Also, parts of the appendicies are omitted.

1

Introduction

The DECsystem-10 is a general purpose, stored program computing system that includes at least one PDP-10 central processor, a memory, and a variety of peripheral equipment such as paper tape reader and punch, teletypewriter, card reader and punch, line printer, DECtape, magnetic tape, disk, drum, display and data communications equipment. Each central processor is the control unit for an entire large-scale subsystem, in which it is connected by an in-out bus to its own peripheral equipment and by a memory bus to one or more memory units in a main memory, some of whose units may be shared by several processors. Within the subsystem the central processor governs all peripheral equipment, sequences the program, and performs all arithmetic, logical and data handling operations. Besides central processors, there are also direct-access processors, which have much more limited program capability and serve to connect large, fast peripheral devices to memory bypassing the central processor. Every direct-access processor is connected to the in-out bus of some central processor, to which it appears as an in-out device; the direct-access processor is also connected to memory by its own memory bus, and to its peripheral equipment by a device bus. The DECsystem-10 may also contain peripheral subsystems, such as for data communications, which are themselves based on small computers; such a subsystem in toto is connected to a PDP-10 in-out bus and is treated by the PDP-10 as a peripheral device. Unless otherwise specified, the words "processor" and "central processor" refer to the large-scale PDP-10 central processor, and "in-out bus" refers to the bus from the central processor to its peripheral equipment. A direct-access processor and the bus to its peripheral equipment are all always referred to by their names, *eg* the DF10 data channel and its channel bus (often a direct-access processor and device control are a single unit).

Confusion could result only in a chapter dealing with a small-computer subsystem. Here the small processor is usually referred to by its name (PDP-8, PDP-11) and the words "computer" and "memory" refer to the small computer. To differentiate, the PDP-10 is referred to by its name or as the "DECsystem-10 central processor", and the large scale memory connected to the PDP-10 memory bus is referred to as "DECsystem-10 main memory".

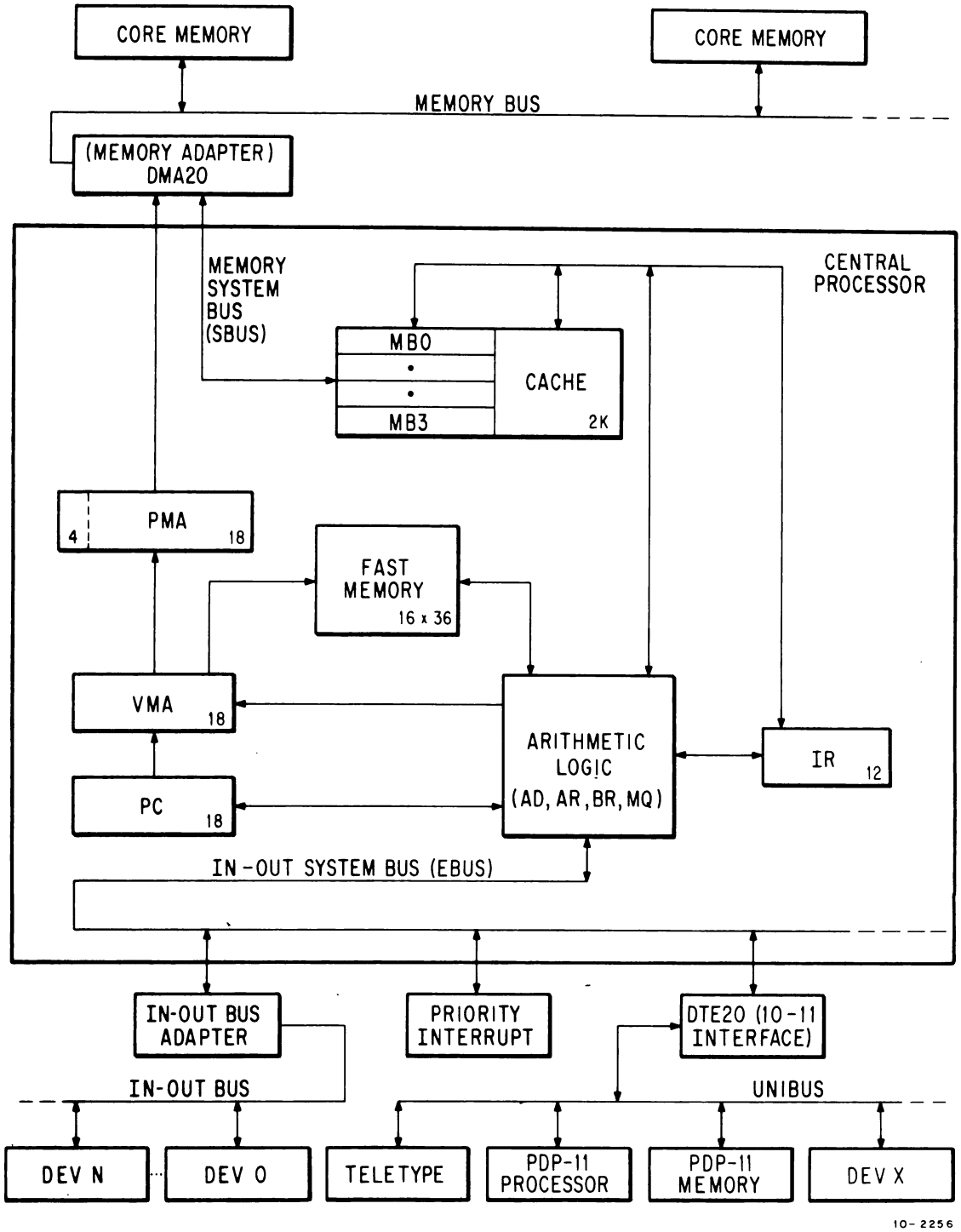
At present, there are three types of PDP-10 central processors, the KA10, the KI10, and the KL10. The three processors handle words of thirty-six bits, which are stored in a memory whose maximum capacity depends upon the addressing capability of the processor. Internally, the processors use 18-bit addresses and can thus reference 262,144 word locations in memory. This is the total addressing capability of the KA10. However, in the KI10 and KL10, it is only the virtual address space available to a single program. Paging hardware supplies four additional address bits to map pages in the program virtual address space into pages anywhere in a physical memory that is sixteen times as large. Thus for a number of different programs, the processor actually has access to a physical memory with a capacity of 4,194,304 words. Storage in memory is usually in the form of 37-bit words,

the extra bit producing odd parity for the word. The bits of a word are numbered 0–35, left to right (most significant to least significant), as are the bits in the registers that handle the words. The processor can handle half words, wherein the left half comprises bits 0–17, the right half, bits 18–35. There is also hardware for byte manipulation – a byte is any contiguous set of bits within a word. KA10 registers that hold addresses have eighteen bits, numbered 18–35 according to the position of an address in a word. The KI10 and KL10 internal address registers have eighteen bits, but a register that must supply a complete address to physical memory has twenty two bits (numbered 14–35). Words are used either as computer instructions in the program, as addresses, or as operands (data for the program).

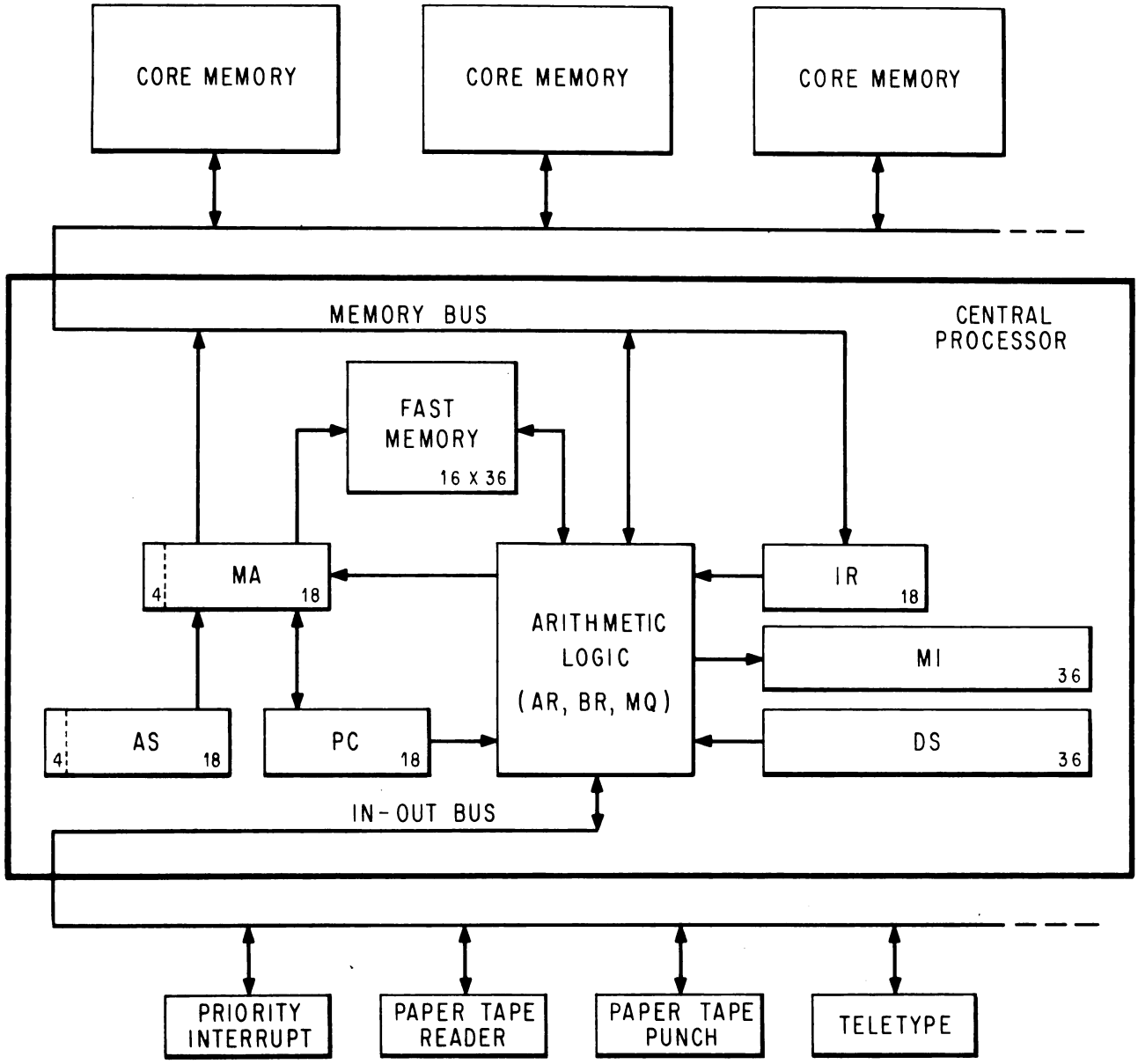
Of the internal registers shown in the illustration on the next page, only PC, the 18-bit program counter, is directly relevant to the programmer. The processor performs a program by executing instructions retrieved from the locations addressed by PC. At the beginning of each instruction PC is incremented by one so that it normally contains an address one greater than the location of the current instruction. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time in a skip instruction or by replacing its contents with the value specified by a jump instruction. Also of importance to the KA10 and KI10 programmer are the sense switches and the 36-bit data switch register DS on the processor console; through these switches the program can read information supplied by the operator. The processor also contains flags that detect various types of errors, including several types of overflow in arithmetic and pushdown operations, and provide other information of interest to the programmer.

The processor has other registers, but the programmer is not usually concerned with them except when manually stepping through a program to debug it. On the KA10 and KI10, the operator can use the address switch register AS, to examine the contents of, or deposit information into, any memory location; stop or interrupt the program whenever a particular location is referenced; and through AS, the operator can supply a starting address for the program. Through the memory indicators MI the program can display data for the operator. In the KL10, these functions are provided by the PDP-11 console, which is connected to the KL10 by a hardware interface. The instruction register IR contains the left half of the current instruction word, *ie,* all but the address part. The memory address register MA (PMA in the KL10) supplies the address for every memory access. The heart of the processor is the arithmetic logic, principally the 36-bit arithmetic register AR. This register takes part in all arithmetic, logical and data handling operations; all data transfers to and from memory, peripheral equipment and console are made via AR. Associated with AR are an extremely fast full adder, a buffer register that holds a second operand in many arithmetic and logical instructions, a multiplier-quotient register MQ that serves primarily as an extension of AR for handling double length operands, and smaller registers that handle floating point exponents and control shift operations and byte manipulation.

From the point of view of the programmer however the arithmetic logic can be regarded as a black box. It performs almost all of the operations



DECSYSTEM-10 SIMPLIFIED, KL10



DECSYSTEM–10 SIMPLIFIED, KA10 AND KI10

necessary for the execution of a program, but it never retains any information from one instruction to the next. Computations performed in the black box either affect control elements such as PC and the flags, or produce results that are always sent to memory and must be retrieved by the processor if they are to be used as operands in other instructions.

An instruction word has only one 18-bit address field for addressing any location throughout all of the virtual address space. But most instructions have two 4-bit fields for addressing the first sixteen memory locations. Any instruction that requires a second operand has an accumulator address field,

which can address one of these sixteen locations as an accumulator; in other words as though it were a result held over in the processor from some previous instruction (the programmer usually has a choice of whether the result of the instruction will go to the location addressed as an accumulator or to that addressed by the 18-bit address field, or to both). Every instruction has a 4-bit index register address field, which can address fifteen of these locations for use as index registers in modifying the 18-bit memory address (a zero index register address specifies no indexing). Although all computations on both operands and addresses are performed in the single arithmetic register AR, the computer actually has sixteen accumulators, fifteen of which can double as index registers. The factor that determines whether one of the first sixteen locations in memory is an accumulator or an index register is not the information it contains nor how its contents are used, but rather how the location is addressed. These first sixteen memory locations are not actually in core memory, but are rather in a fast solid state memory contained in the processor. This allows much quicker access to these locations whether they are addressed as accumulators, index registers or ordinary memory locations. They can even be addressed from the program counter, gaining faster execution for a short but oft-repeated subroutine.

The KI10 actually has four fast memory blocks (eight in the KL10), but only one of these is available to a program at any given time.

Besides the registers that enter into the regular execution of the program and its instructions, the processor has a priority interrupt system and equipment to facilitate time sharing. The interrupt system facilitates processor control of the peripheral equipment by means of a number of priority-ordered channels over which external signals may interrupt the normal program flow. The processor acknowledges an interrupt request by executing the instruction contained in a particular location for the channel or doing some special operation specified by the device (such as incrementing the contents of a memory location). Assignment of channels to devices is entirely under program control. One of the devices to which the program can assign a channel is the processor itself, allowing internal conditions such as overflow or a parity error to signal the program.

Time Sharing. Inherent in the basic machine hardware are restrictions that apply universally: only certain instructions can be used to respond to a priority interrupt, and certain memory locations have predefined uses. But above this fundamental level, the time share hardware provides for different modes of processor operation and establishes certain instruction restrictions and memory restrictions so that the processor can handle a number of user programs (programs run in user mode) without their interfering with one another. The memory restrictions are dependent to a great extent on the processor, but the instruction restrictions are not, and these are relatively obvious: a program that is sharing the system with others cannot usually be allowed to halt the processor or to operate the in-out equipment arbitrarily. A program that runs in executive mode – the Monitor – is responsible for scheduling user programs, servicing interrupts, handling input-output needs, and taking action when control is returned to it from a user program. Any violation of an instruction or memory restriction by a user transfers control back to the Monitor. Dedication of the entire facility to a single purpose, in other words with only one user, is equivalent to

The KI10 and KL10 allow unrestricted in-out with a limited number of devices for special real time applications.

operation in executive mode (specifically kernel mode in the KI10 and KL10).

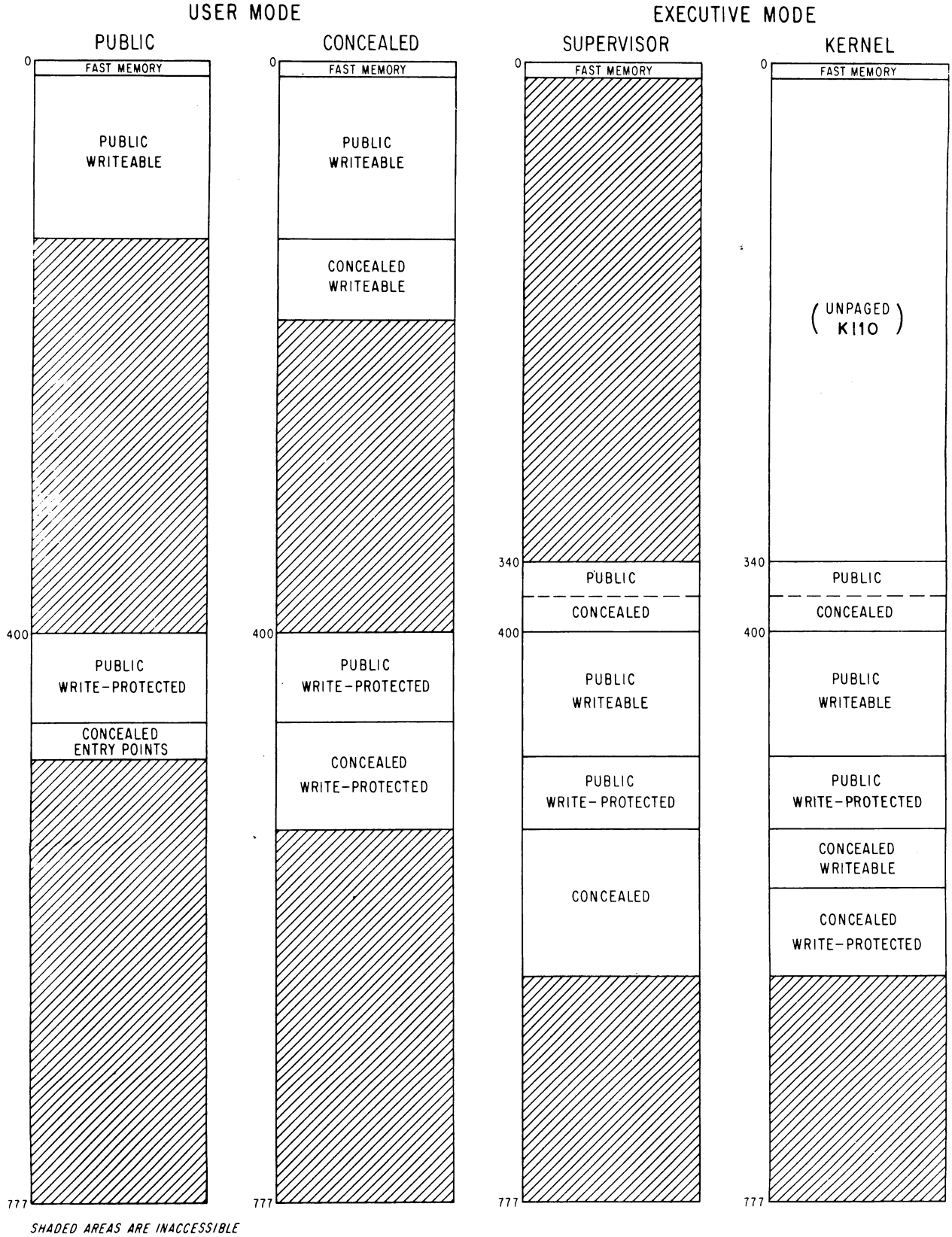
The KA10 has the two modes discussed above, user and executive. It also has protection and relocation hardware to confine the user virtual address space within a particular range, and to relocate user memory references to the appropriate area in physical core. A user ordinarily has access to two separate core areas, one of which may be write-protected, *ie* the user cannot alter its contents.

The KI10 and KL10 have paging hardware for the mapping of pages from the limited virtual address space into pages anywhere in physical memory. A page map for each program specifies not only the correspondence from virtual address to physical address, but also whether an individual page is accessible or not, alterable or not, and public or concealed. Both user and executive modes are subdivided according to whether the program is running in a public area or a concealed area. Within user mode these are the public and concealed modes; within executive mode, the supervisor and kernel modes. A program in concealed mode can reference all of accessible user memory, but the public program cannot reference the concealed area except to transfer control into it at certain legitimate entry points.

The concealed area would ordinarily be used for proprietary programs that the user can call but cannot read or alter.

In kernel mode the Monitor handles the in-out for the system, handles priority interrupts, constructs page maps, and performs those functions that affect all users. This mode has no instruction restrictions and in the KI10 processor the program can even address some of memory directly (*ie* unpaged). In the KL10 processor, the entire executive address space is paged. In the paged address space, individual pages may be restricted as inaccessible or write-protected, but it is the kernel mode program that establishes these restrictions. In supervisor mode, the Monitor handles the general management of the system and those functions that affect only one user at a time. This mode has essentially the same instruction and memory restrictions as user mode, although the supervisor mode program can read, but not alter, the concealed areas; in this way the kernel mode Monitor supplies the supervisor program with information the latter cannot alter (even though the information is not write-protected from the kernel program). In either mode the Monitor automatically uses fast memory block 0 (the hardware requires this). The kernel program is responsible for assigning fast memory blocks to the various user programs: ordinarily blocks 2 and 3 are for special real time applications, and block 1 is assigned to all other users.

The illustration on the next page shows a typical layout of the virtual address space for the various modes. The space is 256K, made up of 512 pages numbered 0–777 octal. Any program can address locations 0–17 as these are in a fast memory block and are completely unrestricted (although the same addresses may be in different blocks for different programs). The public mode user program operates in the public area, part of which may be write-protected. The public program cannot access any locations in the concealed areas except to fetch instructions from prescribed entry points. The concealed mode user program has access to both public and concealed areas, but it cannot alter any write-protected location whether public or concealed, and fetching an instruction from the public area automatically returns the processor to public mode.



TYPICAL VIRTUAL ADDRESS SPACE CONFIGURATION

In the KI10 only, the supervisor mode program is confined within the paged area of the address space, pages 340 and above. Part of the public area in this space may be write-protected, but the program can read information in the concealed areas – it cannot alter any location in a concealed area whether that area is write-protected or not. Pages 340–377 constitute the per-process area, which contains information specific to individual users and whose mapping accompanies the user page map. In other words the physical memory corresponding to these virtual pages can be changed simply by switching from one user to another, rather than the Monitor changing its own page map. The kernel mode program can access all of the unpaged area without restriction and can reference all of the accessible paged area, both public and concealed, with the usual restriction that it cannot alter a write-protected area. As in the case of concealed user mode, fetching an instruction from a public area returns control to supervisor mode.

In the KI10, the kernel address space (low 112K) is unpaged. In the KL10, the entire kernel address space is normally paged.

1.1 NUMBER SYSTEM

The program can interpret a data word as a 36-digit, unsigned binary number, or the left and right halves of a word can be taken as separate 18-bit numbers. The PDP-10 repertoire includes instructions that effectively add or subtract one from both halves of a word, so the right half can be used for address modification when the word is addressed as an index register, while the left half is used to keep a control count.

The standard arithmetic instructions in the PDP-10 use twos complement, fixed point conventions to do binary arithmetic. In a word used as a number, bit 0 (the leftmost bit) represents the sign, 0 for positive, 1 for negative. In a positive number the remaining 35 bits are the magnitude in ordinary binary notation. The negative of a number is obtained by taking its twos complement. If x is an n-digit binary number, its twos complement is $2^n - x$, and its ones complement is $(2^n - 1) - x$, or equivalently $(2^n - x) - 1$. Subtracting a number from $2^n - 1$ (*ie*, from all 1s) is equivalent to performing the logical complement, *ie* changing all 0s to 1s and all 1s to 0s. Therefore, to form the twos complement one takes the logical complement (usually referred to merely as the complement) of the entire word including the sign, and adds 1 to the result. In a negative number the sign bit is 1, and the remaining bits are the twos complement of the magnitude.

The adder actually acts as though the words represented 36-bit unsigned numbers, *ie* the signs are treated just like magnitude bits. In the absence of a carry into the sign stage, adding two numbers with the same sign produces a plus sign in the result. The presence of a carry gives a positive answer when the summands have different signs. The result has a minus sign when there is a carry into the sign bit and the summands have the same sign, or the summands have different signs and there is no carry.

Thus the program can interpret the numbers processed in fixed point addition and subtraction as signed numbers with 35 magnitude bits or as unsigned 36-bit numbers. A computation on signed numbers produces a result that

$+153_{10} = +231_8 =$ 000 000 000 000 000 000 000 000 000 010 011 001

0 35

$-153_{10} = -231_8 =$ 111 111 111 111 111 111 111 111 111 101 100 111

0 35

Zero is represented by a word containing all 0s. Complementing this number produces all 1s, and adding 1 to that produces all 0s again. Hence there is only one zero representation and its sign is positive. Since the numbers are symmetrical in magnitude about a single zero representation, all even numbers both positive and negative end in 0, all odd numbers in 1 (a

number all 1s represents −1). But since there are the same number of positive and negative numbers and zero is positive, there is one more negative number than there are nonzero positive numbers. This is the most negative number and it cannot be produced by negating any positive number (its octal representation is $400000\,000000_8$ and its magnitude is one greater than the largest positive number).

If ones complements were used for negatives one could read a negative number by attaching significance to the 0s instead of the 1s. In twos complement notation each negative number is one greater than the complement of the positive number of the same magnitude, so one can read a negative number by attaching significance to the rightmost 1 and attaching significance to the 0s at the left of it (the negative number of largest magnitude has a 1 in only the sign position). In a negative integer, 1s may be discarded at the left, just as leading 0s may be dropped in a positive integer. In a negative fraction, 0s may be discarded at the right. So long as only 0s are discarded, the number remains in twos complement form because it still has a 1 that possesses significance; but if a portion including the rightmost 1 is discarded, the remaining part of the fraction is now a ones complement.

is correct as an unsigned 36-bit number even if overflow occurs, but the hardware interprets the result as a signed number to detect overflow. Adding two positive numbers whose sum is greater than or equal to 2^{35} gives a negative result, indicating overflow; but that result, which has a 1 in the sign bit, is the correct answer interpreted as a 36-bit unsigned number in positive form. Similarly adding two negatives gives a result which is always correct as an unsigned number in negative form.

The computer does not keep track of a binary point – the programmer must adopt a point convention and shift the magnitude of the result to conform to the convention used. Two common conventions are to regard a number as an integer (binary point at the right) or as a proper fraction (binary point at the left); in these two cases the range of numbers represented by a single word is -2^{35} to $2^{35}-1$ or -1 to $1-2^{-35}$. Since multiplication and division make use of double length numbers, there are special instructions for performing these operations with integral operands.

Multiplication produces a double length product, and the programmer must remember that discarding the low order part of a double length negative leaves the high order part in correct twos complement form only if the low order part is null.

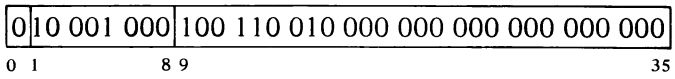
The format for double length fixed point numbers is just an extension of the single length format. The magnitude (or its twos complement) is the 70-bit string in bits 1–35 of the high and low order words. Bit 0 of the high order word is the sign, and bit 0 of the low order word is made equal to the sign. The range for double length integers and proper fractions is thus -2^{70} to $2^{70}-1$ and -1 to $1-2^{-70}$.

▲ This convention for bit 0 of the low order word is inconsistent with that used for floating point arithmetic [*see below*]. This should cause no problem however, as fixed divide ignores bit 0 of the low order word in a double length dividend.

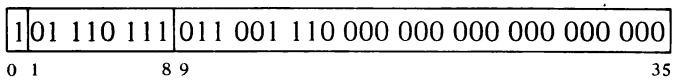
Floating Point Arithmetic. The KI10 and KL10 have hardware for processing single and double precision floating point numbers; the KA10 can generally process only single precision numbers, although the hardware does include features that facilitate double precision arithmetic by software routines. The same format is used for a single precision number and the high order word of a double precision number. A floating point instruction interprets bit 0 as the sign, but interprets the rest of the word as an 8-bit exponent and a 27-bit fraction. For a positive number the sign is 0, as before. But the contents of bits 9–35 are now interpreted only as a binary fraction, and the contents of bits 1–8 are interpreted as an integral exponent in excess 128 (200_8) code. Exponents from −128 to +127 are therefore represented by the binary equivalents of 0 to 255 ($0-377_8$). Floating point zero and negatives are represented in exactly the same way as in fixed point: zero by a word containing all 0s, a negative by the twos complement. A negative number has a 1 for its sign and the twos complement of the fraction, but since every fraction must ordinarily contain a 1 unless the entire number is zero (see below), it has the ones complement of the exponent code in bits 1–8. Since the exponent is in excess 128 code, an

actual exponent x is represented in a positive number by x +128, in a negative number by 127– x. The programmer, however, need not be concerned with these representations as the hardware compensates automatically. *Eg,* for the instruction that scales the exponent, the hardware interprets the integral scale factor in standard twos complement form but produces the correct ones complement result for the exponent.

$+153_{10} = +231_8 = +.462_8 \times 2^8 =$



$-153_{10} = -231_8 = -.462_8 \times 2^8 =$



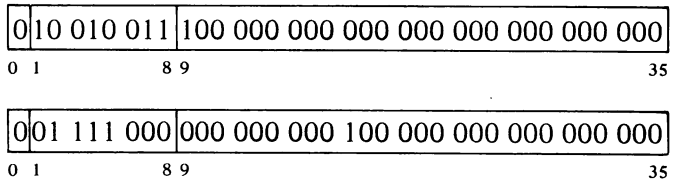
Except in special cases the floating point instructions assume that all nonzero operands are normalized, and they normalize a nonzero result. A floating point number is considered normalized if the magnitude of the fraction is greater than or equal to ½ and less than 1. The hardware may not give the correct result if the program supplies an operand that is not normalized or that has a zero fraction with a nonzero exponent.

Single precision floating point numbers have a fractional range in magnitude of ½ to $1 - 2^{-27}$. Increasing the length of a number to two words does not significantly change the range but rather increases the precision; in any format the magnitude range of the fraction is ½ to 1 decreased by the value of the least significant bit. In all formats the exponent range is −128 to +127.

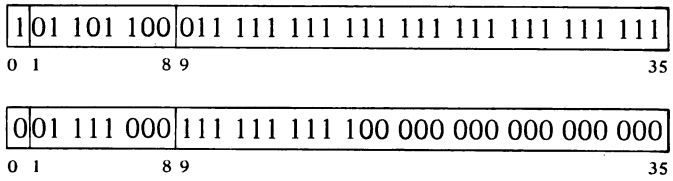
The precaution about truncation given for fixed point multiplication applies to most floating point operations as they produce extra length results; but here the programmer may request rounding, which automatically restores the high order part to twos complement form if it is negative. In single precision division the two words of the result are quotient and remainder, but in the other operations they form a double length number which is stored in two accumulators if the instruction is executed in "long" mode. (Long mode division uses a double length dividend.) A double length number used by the single precision instructions is in software double precision format. As such it contains a 54-bit fraction, half of which is in bits 9–35 of each word. The sign and exponent are in bits 0 and 1–8 respectively of the word containing the more significant half, and the standard twos complement is used to form the negative of the entire 63-bit string. In the remaining part of the less significant word, bit 0 is 0, and bits 1–8 contain a number 27 less than the exponent, but this is expressed in positive form even though bits 9–35 may be part of a negative fraction. *Eg* the number $2^{18} + 2^{-18}$ has this two-word representation in software

An instruction that generates a double length result sets the low word exponent part to zero whenever the low order fraction is zero, and sets the whole low order word to zero whenever the low order exponent overflows or underflows.

double precision format:



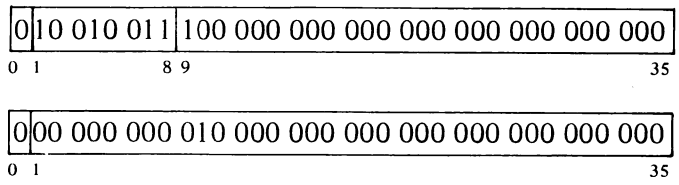
whereas its negative is



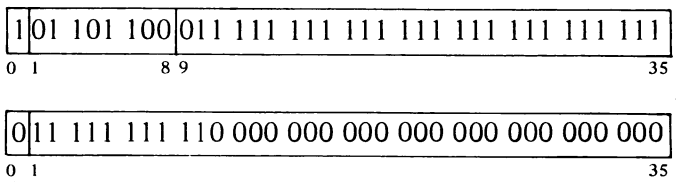
Essentially there are five number formats. Fixed point additive operations can be regarded as being performed on 36-bit unsigned numbers, which are equivalent to logical words. Otherwise fixed point arithmetic uses the fixed point format; numbers are single length with the exception that products and dividends can be double length, and there is provision for shifting a double length operand arithmetically. Double length format is an extension of single length format to two 36-bit words.

Single precision floating point instructions use two formats: single precision floating point format and *software* double precision floating point format. The latter appears only in the result of a long mode add, subtract or multiply, as the dividend in a long mode divide, and as the operand for an instruction that negates a number specifically in that format. Operands for double precision floating point instructions are exclusively in *hardware* double precision floating point format (and these instructions are not available on the KA10).

The double precision floating point instructions use a more straightforward double length format with greater precision than is allowed by the software format. For these instructions all operands and results are double length, and all instructions except division calculate a triple length answer, which is rounded to double length with the appropriate adjustment for a twos complement negative. In hardware double precision format the high order word is the same as a single precision number, and bits 1–35 of the low order word are simply an extension of the fraction, which is now sixty-two bits. Bit 0 is ignored. The number used above as an example of software double precision format has this representation in hardware format:



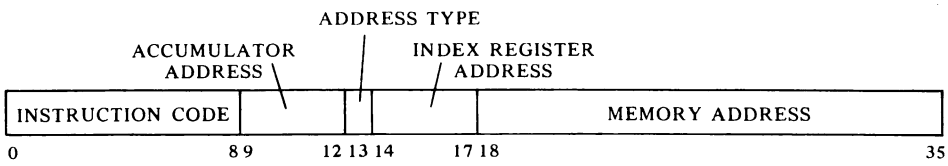
and its negative is



1.2 INSTRUCTION FORMAT

In all but the input-output instructions, the nine high order bits (0–8) specify the operation, and bits 9–12 usually address an accumulator but are sometimes used for special control purposes, such as addressing flags. The

rest of the instruction word usually supplies information for calculating the effective address, which is the actual address used to fetch the operand or alter program flow. Bit 13 specifies the type of addressing, bits 14–17 specify an index register for use in address modification, and the remaining eighteen bits (18–35) address a memory location. The instruction codes

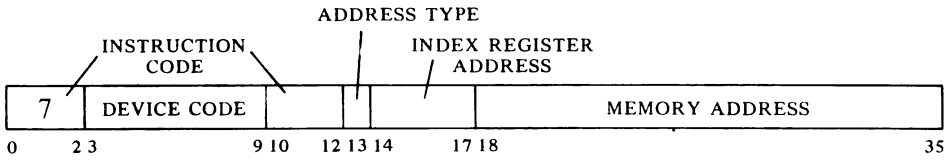


BASIC INSTRUCTION FORMAT

that are not assigned as specific instructions are performed by the processor as so-called "unimplemented operations".

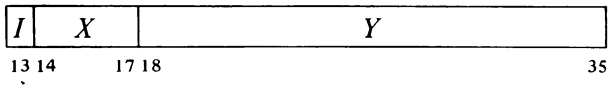
An input-output instruction is designated by three 1s in bits 0–2. Bits 3–9 address the in-out device to be used in executing the instruction, and bits 10–12 specify the operation. The rest of the word is the same as in other instructions.

Among the unimplemented operations are some that are specified as "unimplemented user operations" or UUOs (a mnemonic that means nothing to the assembler). Half of these are for the local use of a program (LUUOs) and the other half are for communication with the Monitor (MUUOs). In general, unassigned codes act like MUUOs.



IN-OUT INSTRUCTION FORMAT

Effective Address Calculation. Bits 13–35 have the same format in *every* instruction whether it addresses a memory location or not. Bit 13 is the



indirect bit, bits 14–17 are the index register address, and if the instruction must reference memory, bits 18–35 are the memory address Y. The effective address E of the instruction depends on the values of I, X and Y. If X is nonzero, the contents of index register X are added to Y to produce a modified address. If I is 0, addressing is direct, and the modified address is the effective address used in the execution of the instruction; if I is 1, addressing is indirect, and the processor retrieves another address word from the location specified by the modified address already determined. This new word is processed in exactly the same manner: X and Y determine the effective address if I is 0, otherwise they are used for yet another level of address retrieval. This process continues until some referenced location is found with a 0 in bit 13; the 18-bit number calculated from the X and Y parts of this location is the effective address E.

The calculation outlined above is carried out for *every* instruction even if it need not address a memory location. If the indirect bit in the instruc-

On the other hand, please note that this calculation is carried

out only for words indicated in the text as having the format shown. Do not assume that the procedure is used for any miscellaneous pointer simply because it happens to contain an address [*see page C-2*].

tion word is 0 and no memory reference is necessary, then *Y* is not an address. It may be a mask in some kind of test instruction, conditions to be sent to an in-out device, or part of it may be the number of places to shift in a shift or rotate instruction or the scale factor in a floating scale instruction. Even when modified by an index register, bits 18–35 do not contain an address when *I* is 0. But when *I* is 1, the number determined from bits 14–35 is an indirect address no matter what type of information the instruction requires, and the word retrieved in any step of the calculation contains an indirect address so long as *I* remains 1. When a location is found in which *I* is 0, bits 18–35 (perhaps modified by an index register) contain the desired effective mask, effective conditions, effective shift number, or effective scale factor. Many of the instructions that usually reference memory for an operand even have an "immediate" mode in which the result of the effective address calculation is itself used as a half word operand instead of a word taken from the memory location it addresses.

The important thing for the programmer to remember is that the same calculation is carried out for every instruction regardless of the type of information that must be specified for its execution, or even if the result is ignored. In the discussion of any instruction, *E* refers to the actual quantity derived from *I*, *X* and *Y* and used in the execution of the instruction, be it the entire half word as in the case of an address, immediate operand, mask or conditions, or only part of it as in a shift number or scale factor.

PLEASE READ THIS

The calculation of E is the first step in the execution of every instruction. No other action taken by any instruction, no matter what it is, can possibly precede that calculation. There is absolutely nothing whatsoever that any instruction can do to any accumulator or memory location that can in any way affect its own effective address calculation.

1.3 MEMORY

The internal timing for each in-out device and each memory is entirely independent of the central processor. Because core memory readout is destructive, every word read must be written back in unless new information is to take its place. But the processor need never wait the entire cycle time. To read, it waits only until the information is available and then continues its operations while the memory performs the write portion of the cycle; to write, it waits only until the data is accepted, and the memory then performs an entire cycle to clear and write. To save time in an instruction that fetches an operand and then writes new data into the same location, the memory executes a read-modify-write cycle in which it performs only the read part initially and then completes the cycle when the processor supplies the new data. This procedure is not used however in a lengthy instruction (such as multiply or divide), which would tie up a memory that may be needed by some other processor. Such instructions instead request separate read and write access. The KI10 further increases the speed of memory operation by overlapping memory cycles. *Eg* it can start one memory to read a word before receiving a word previously requested from a different memory.

The KL10 contains a high-speed cache which holds some selection of words from the main memory system. This reduces access time and the percentage of main memory cycles required by the central processor.

Access times for the accumulator-index register locations are decreased considerably by substitution of a fast memory (contained in the processor) for the first sixteen core locations. Readout is nondestructive, so the fast memory has no basic cycle: the processor reads or writes a word directly (note: to write, the KA10 must first clear the location and then load it).

The following table gives the characteristics of the various memories. Modify completion is the time to finish a read-modify-write cycle after the processor supplies the new data. Times are in microseconds and include the delay introduced by ten feet (three meters) of cable. Fast memory times are for referencing as a memory location (18-bit address); when a fast memory location is addressed as an accumulator or index register, the access time is considerably shorter.

	Read Access	*Write Access*	*Cycle*	*Modify Completion*	*Size*
161 Core Memory	2.5	.49	4.7	2.69	16K
163 Core Memory	.94	.49	1.8	1.33	16K
164 Core Memory MB10 Core Memory	.60*	.20*	1.65*	.97	16K
MA10 Core Memory	.61	.20	1.00	.57	16K
MD10 Core Memory	.83	.33	1.8	1.23	32–128K
ME10 Core Memory	.61	.20	1.00	.65	16K
MF10 Core Memory	.61	.20	1.00	.63	32K,64K
MG10 Core Memory	.67	.23	1.00		32–128K
KA10 Fast Memory	.21	.21			16K
KI10 Fast Memory	.28	.0			16K
KL10 Fast Memory	.12	.08			16K
KL10 Cache Memory	.16	.12			2K

*Add .1 in a multiprocessor system.

MD10 can be increased in units of 32K up to 128K.

KI10 access to accumulators and index registers effectively takes no time – it is done in parallel with instruction operations that are required anyway. Retrieval of instructions or memory operands from fast memory is generally not worthwhile because of the extensive overlapping that speeds up core access. However, except in instructions that use two accumulators, storage of a memory operand in fast memory not only takes no time but actually decreases slightly the nonmemory time.

From the simple hardware addressing point of view, the entire memory is a set of contiguous locations whose addresses range from zero to a maximum dependent upon the capacity of the particular installation. In a system with the greatest possible capacity, the largest KA10 address is octal 777777, decimal 262,143; the largest KI10 or KL10 address is 17777777, decimal 4,194,303. (Addresses are always in octal notation unless otherwise specified.) But the whole memory would usually be made up of a number of core memories of different capacities as listed above. Hence a given address actually selects a particular memory and a specific location within it. For a 16K memory with 18-bit addressing, the high order four address bits select the memory, the remaining fourteen bits address a single location in it; selecting a 32K memory takes three bits, leaving fifteen for the location. The times given above assume the addressed memory is idle when access is requested. To avoid waiting for a previously requested memory cycle to end, the program can make consecutive requests to different memories by taking instructions from one memory and data from another. All memories can be interleaved in pairs i such a way that consecutive addresses actually alternate between the two memories in the pair (thus increasing the probability that consecutive references are to different memories). Appropriate switch settings at the memories interchange the least significant address bits in the memory selection and location parts, so that in any two memories numbered n and n+1 where n is even, all even addresses are locations in the first memory, all odd addresses are locations in the second. Hence memories 0 and 1 can be interleaved as can 6 and 7, but not 3 and 4 or 5 and 7. Some

Information on memory setup is given in Appendix G.

memories can be interleaved in contiguous groups of four, where the number of the first memory in the group is divisible by four (*eg* memories 0–3 or 14–17). In this case all addresses ending in 0 or 4 reference the first memory in the group, all ending in 1 or 5 reference the second, and so forth.

In terms of the virtual address space (the addresses that can be specified within the limits of the instruction format) or the subset of it that is accessible to a user, the situation may be quite different. In the KA10 the user program has a continuous address space beginning at 0, or two continuous spaces beginning at 0 and 400000. In the KI10 the possible program address space is the set of all 18-bit addresses just as in the KA10, but which addresses a program can actually use depends entirely upon which of the 512 virtual pages (512 words per page) are accessible to it. For a so-called "small user", the accessible space must lie within the ranges 0–37777 and 400000–437777. In any event all programs have access to fast memory, whether as accumulators, index registers or ordinary memory references (*ie* addresses 0–17 are never restricted or relocated).

In the KI10, the kernel mode program can always address locations 0-337777 as these are unpaged. Virtual pages 340 and above are mapped.

KI10 Memory Allocation. The KI10 hardware defines the use of certain memory locations, but most are relative to pages whose physical location is specified by the Monitor. The auto restart uses location 70. The only other physical locations uniquely defined by the hardware are those in fast memory, whose addresses are the same for all programs: location 0 holds a pointer word during a bootstrap readin, 0–17 can be addressed as accumulators, and 1–17 can be addressed as index registers. The only addresses uniquely specified in the user virtual space are for user local UUOs – locations 40 and 41.

All other addresses defined by the hardware, for use in page mapping, responding to priority interrupts, or other hardware-oriented situations, are to locations within a page specified by the Monitor for a particular user (including itself). For each user the Monitor keeps a process table, which must begin at location 0 of some page. The locations used by the hardware for the page map, traps, etc. of a given user are all in the first page of the table for that user. The parts of a user process table not used by the hardware may be used by the Monitor to keep accumulators (when the user is not running), a pushdown list that the Monitor uses for the job, and various user statistics such as running time, memory space, billing information, and job tables. The detailed configuration of the hardware-defined parts of the process tables (user and executive) is given in §2.15.

The Monitor keeps a user process table for each user program and one executive process table for itself for each KI10 processor. In the text, the phrase "the user process table" refers to the process table currently specified by the Monitor as the one for the user, even if that user is not currently running. The Monitor must also specify the whereabouts of the executive process table for the processor under consideration.

KA10 Memory Allocation. The use of certain memory locations is defined by the KA10 hardware.

0	Holds a pointer word during a bootstrap readin
0–17	Can be addressed as accumulators
1–17	Can be addressed as index registers
40–41	Trap for unimplemented user operations (UUOs)
42–57	Priority interrupt locations
60–61	Trap for remaining unimplemented operations: these include the unassigned instruction codes that are reserved for future use, and also the byte manipulation and floating point instructions when the hardware for them is not installed

The initial control word address for the DF10 Data Channel must be less than 1000.

140–161 Allocated to second processor if connected (same use as 40–61 for first processor)

All information given in this manual about memory locations 40–61 for a KA10 applies instead to locations 140–161 for programming a second KA10 connected to the same memory.

In a user program the trap for a local UUO is relocated to locations 40 and 41 of the user area; a Monitor UUO uses unrelocated locations. All other addresses listed are for physical (unrelocated) locations.

1.4 PROGRAMMING CONVENTIONS

The computer has five instruction classes: data transmission, logical, arithmetic, program control and in-out. The instructions in the in-out class control the peripheral equipment, and also control the priority interrupt and time sharing, control and read the processor flags, and communicate with the console. The next chapter describes all instructions mentioned above, presents a general description of input-output, and describes the effects of the in-out instructions on the processor, priority interrupt and time share hardware. Effects of in-out instructions on particular peripheral devices are discussed with the devices.

The MACRO–10 assembly program recognizes a number of mnemonics and other initial symbols that facilitate constructing complete instruction words and organizing them into a program. In particular there are mnemonics for the instruction codes (Appendix A), which are six bits in in-out instructions, otherwise nine or thirteen bits. *Eg* the mnemonic

```
MOVNS
```

The assembler translates every statement into a 36-bit word, placing 0s in all bits whose values are unspecified.

assembles as 213000 000000, and

```
MOVNS  2570
```

assembles as 213000 002570. This latter word, when executed as an instruction, produces the twos complement negative of the word in memory location 2570.

NOTE

Throughout this manual all numbers representing instruction words, register contents, codes and addresses are always octal, and any numbers appearing in program examples are octal unless otherwise indicated. On the other hand, the ordinary use of numbers in the text to count steps in an operation or to specify word or byte lengths, bit positions, exponents, etc employs standard decimal notation.

The initial symbol @ preceding a memory address places a 1 in bit 13 to produce indirect addressing. The example given above uses direct addressing, but

```
MOVNS  @2570
```

assembles as 213020 002570, and produces indirect addressing. Placing the

number of an index register (1–17) in parentheses following the memory address causes modification of the address by the contents of the specified register. Hence

```
MOVNS  @2570(12)
```

which assembles as 213032 002570, produces indexing using index register 12, and the processor then uses the modified address to continue the effective address calculation.

An accumulator address (0–17) precedes the memory address part (if any) and is terminated by a comma. Thus

```
MOVNS  4,@2570(12)
```

assembles as 213232 002570, which negates the word in location *E* and stores the result in both *E* and in accumulator 4. The same procedure may be used to place 1s in bits 9–12 when these are used for something other than addressing an accumulator, but mnemonics are available for this purpose.

The device code in an in-out instruction is given in the same manner as an accumulator address (terminated by a comma and preceding the address part), but the number given must correspond to the octal digits in the word (000–774). Mnemonics are however available for all standard device codes. To control the priority interrupt system whose code is 004, one may give

```
CONO   4,1302
```

which assembles as 700600 001302, or equivalently

```
CONO   PI,1302
```

The programming examples in this manual use the following addressing conventions:

♦ A colon following a symbol indicates that it is a symbolic location name.

```
A:      ADD    6,5704
```

indicates that the location that contains ADD 6,5704 may be addressed symbolically as A.

♦ The period represents the current address, *eg*

```
        ADD    5,.+2
```

is equivalent to

```
A:      ADD    5,A+2
```

♦ Square brackets specify the contents of a location, leaving the address of the location implicit but unspecified. *Eg*

```
        ADD    12,[7256004]
```

and

```
        ADD    12,A
```

```
        .
        .
        .
A:      7256004
```

are equivalent. The bracketed quantity can be given as the left and right halves separated by a double comma, not only eliminating the need to insert leading zeros for the right half, but allowing use of a minus sign for a negative half word as well. In other words

```
[-246,,135]
```

is equivalent to

```
[777532000135]
```

Anything written at the right of a semicolon is commentary that explains the program but is not part of it.

2
Central Processor

This chapter describes all PDP-10 instructions but does not discuss the effects of those in-out instructions that address specific peripheral devices. In the description of each instruction, the mnemonic and name are at the top, the format is in a box below them. The mnemonic assembles to the word in the box, where bits in those parts of the word represented by letters assemble as 0s. The letters indicate portions that must be added to the mnemonic to produce a complete instruction word.

For many of the non-IO instructions, a description applies not to a unique instruction with a single code in bits 0-8, but rather to an instruction set defined as a basic instruction that can be executed in a number of modes. These modes define properties subsidiary to the basic operation; *eg* in data transmission the mode specifies which of the locations addressed by the instruction is the source and which the destination of the data, in test instructions it specifies the condition that must be satisfied for a jump or skip to take place. The mnemonic given at the top is for the basic mode; mnemonics for the other forms of the instruction are produced by appending letters directly to the basic mnemonic. Following the description is a table giving the mnemonics and octal codes (bits 0-8) for the various modes.

Letters representing modes are suffixes, which produce new mnemonics that are recognized as distinct symbols by the assembler.

In a description *E* refers to the effective address, half word operand, mask, conditions, shift number or scale factor calculated from the *I*, *X* and *Y* parts of the instruction word. In an instruction that ordinarily references memory, a reference to *E* as the source of information means that the instruction retrieves the word contained in location *E*; as a destination it means the instruction stores a word in location *E*. In the immediate mode of these instructions, the effective half word operand is usually treated as a full word that contains *E* in one half and zero in the other, and is represented either as 0,*E* or *E*,0 depending upon whether *E* is in the right or left half.

Most of the non-IO instructions can address an accumulator, and in the box showing the format this address is represented by *A*; in the description, "AC" refers to the accumulator addressed by *A*. "AC left" and "AC right" refer to the two halves of AC. If an instruction uses two accumulators, these have addresses *A* and *A*+1, where the second address is 0 if *A* is 17. In some cases an instruction uses an accumulator only if *A* is nonzero: a zero address in bits 9-12 specifies no accumulator.

The instructions are described in terms of their effects as seen by the user in a normal program situation, and on the assumption that nothing is amiss – the program is not attempting to reference a memory that does not exist or to write in a protected area of core. In general, all descriptions apply equally

PLEASE READ THIS

The calculation of E is the first step in the execution of every instruction. No other action taken by any instruction, no matter what it is, can possibly precede that calculation. There is absolutely nothing whatsoever that any instruction can do to any accumulator or memory location that can in any way affect its own effective address calculation.

well to operation in executive mode. For completeness, the effects of restrictions on certain instructions are noted, as are the effects of executing instructions in special circumstances. But for the details of programming in such special situations the reader must look elsewhere. In particular, §2.9 discusses trapping, §2.13 describes the priority interrupt, and §§2.15, 2.16, and 2.17 describe the special effects and restrictions associated with program and memory management in the KL10, KI10, and KA respectively.

To minimize processor execution time the programmer should minimize the number of memory references and the number of shifts and other iterative operations. When there is a choice of actions to be taken on the basis of some test, the conditions tested should be set up so that the action that results most often takes the least time. There are also various subtleties that affect timing (such as the nature of the arithmetic algorithms), but these are generally not worth considering except in very special circumstances (to determine the effect often takes more than the time saved).

No execution times are given with the instruction descriptions as the time may vary greatly depending upon circumstances. At the outset the time depends upon which processor performs the instruction, the mode the processor is in, and the speeds of the memories used for fetching the instruction, fetching its operands, and storing its results. Beyond this the time depends in many cases on the configuration of the operands and the number of iterative steps specified by the programmer as in a shift. Lastly the processor is designed to save time wherever possible by inspecting the operands in order to skip unnecessary steps.

The text sometimes refers to an instruction as being "executed." To "execute" an instruction means that the processor performs the instruction out of the normal sequence, *ie* the sequence defined by the program counter (which sequence may not be consecutive, as when a skip or jump or some special circumstance changes PC). The processor fetches an executed instruction from a location whose address is supplied not by PC, but rather by an execute instruction (whose operand is itself interpreted as an instruction) or by some feature of the hardware such as a priority interrupt, trap, etc. It is assumed that control will shortly be returned to PC, at the location it originally specified before the interruption unless the instruction executed or the hardware feature itself changes PC.

Some simple examples are included with the instruction descriptions, but more complex examples using a variety of instructions are given in §2.11.

2.1 HALF WORD DATA TRANSMISSION

These instructions move a half word and may modify the contents of the other half of the destination location. There are sixteen instructions determined by which half of the source word is moved to which half of the destination, and by which of four possible operations is performed on the other

half of the destination. The basic mnemonics are three letters that indicate the transfer

HLL	Left half of source to left half of destination
HRL	Right half of source to left half of destination
HRR	Right half of source to right half of destination
HLR	Left half of source to right half of destination

plus a fourth, if necessary, to indicate the operation.

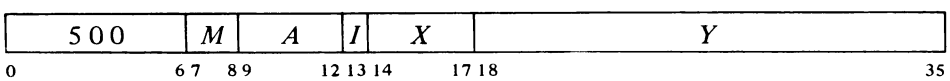
Operation	*Suffix*	*Effect on Other Half of Destination*
Do nothing		None
Zeros	Z	Places 0s in all bits of the other half
Ones	O	Places 1s in all bits of the other half
Extend	E	Places the sign (the leftmost bit) of the half word moved in all bits of the other half. This action extends a right half word number into a full word number but is valid arithmetically only for positive left half word numbers – the right extension of a number requires 0s regardless of sign (hence the Zeros operation should be used to extend a left half word number).

An additional letter may be appended to indicate the mode, which determines the source and destination of the half word moved.

Mode	*Suffix*	*Source*	*Destination*
Basic		*E*	AC
Immediate	I	The word 0, *E*	AC
Memory	M	AC	*E*
Self	S	*E*	*E*, but full word result also goes to AC if *A* is nonzero

Note that selecting the left half of the source in immediate mode merely clears the selected half of the destination.

HLL **Half Word Left to Left**

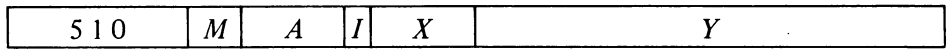


Move the left half of the source word specified by *M* to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination left half are lost.

HLL	Half Left to Left	500
HLLI	Half Left to Left Immediate	501
HLLM	Half Left to Left Memory	502
HLLS	Half Left to Left Self	503

HLLI merely clears AC left. If *A* is zero, HLLS is a no-op, otherwise it is equivalent to MOVE.

HLLZ Half Word Left to Left, Zeros

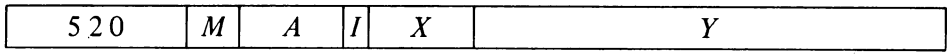


Move the left half of the source word specified by *M* to the left half of the specified destination, and clear the destination right half. The source is unaffected, the original contents of the destination are lost.

HLLZ	Half Left to Left, Zeros	510
HLLZI	Half Left to Left, Zeros, Immediate	511
HLLZM	Half Left to Left, Zeros, Memory	512
HLLZS	Half Left to Left, Zeros, Self	513

HLLZI merely clears AC. If *A* is zero, HLLZS merely clears the right half of location *E*.

HLLO Half Word Left to Left, Ones

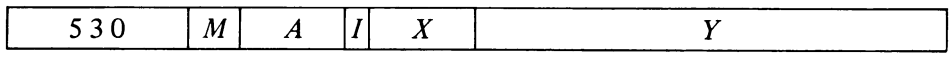


Move the left half of the source word specified by *M* to the left half of the specified destination, and set the destination right half to all 1s. The source is unaffected, the original contents of the destination are lost.

HLLO	Half Left to Left, Ones	520
HLLOI	Half Left to Left, Ones, Immediate	521
HLLOM	Half Left to Left, Ones, Memory	522
HLLOS	Half Left to Left, Ones, Self	523

HLLOI sets AC to all 0s in the left half, all 1s in the right.

HLLE Half Word Left to Left, Extend

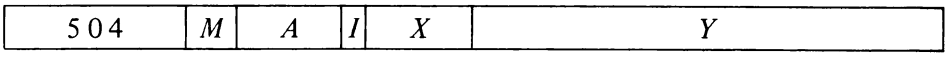


Move the left half of the source word specified by *M* to the left half of the specified destination, and make all bits in the destination right half equal to bit 0 of the source. The source is unaffected, the original contents of the destination are lost.

HLLE	Half Left to Left, Extend	530
HLLEI	Half Left to Left, Extend, Immediate	531
HLLEM	Half Left to Left, Extend, Memory	532
HLLES	Half Left to Left, Extend, Self	533

HLLEI is equivalent to HLLZI (it merely clears AC).

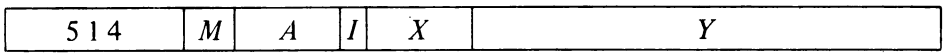
HRL Half Word Right to Left



Move the right half of the source word specified by *M* to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination left half are lost.

HRL	Half Right to Left	504
HRLI	Half Right to Left Immediate	505
HRLM	Half Right to Left Memory	506
HRLS	Half Right to Left Self	507

HRLZ Half Word Right to Left, Zeros

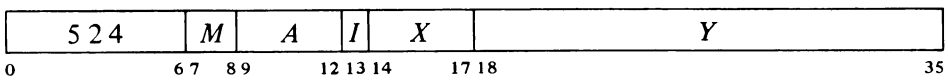


Move the right half of the source word specified by *M* to the left half of the specified destination, and clear the destination right half. The source is unaffected, the original contents of the destination are lost.

HRLZ	Half Right to Left, Zeros	514
HRLZI	Half Right to Left, Zeros, Immediate	515
HRLZM	Half Right to Left, Zeros, Memory	516
HRLZS	Half Right to Left, Zeros, Self	517

HRLZI loads the word *E*,0 into AC.

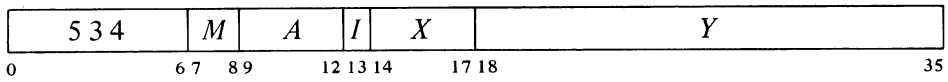
HRLO Half Word Right to Left, Ones



Move the right half of the source word specified by *M* to the left half of the specified destination, and set the destination right half to all 1s. The source is unaffected, the original contents of the destination are lost.

HRLO	Half Right to Left, Ones	524
HRLOI	Half Right to Left, Ones, Immediate	525
HRLOM	Half Right to Left, Ones, Memory	526
HRLOS	Half Right to Left, Ones, Self	527

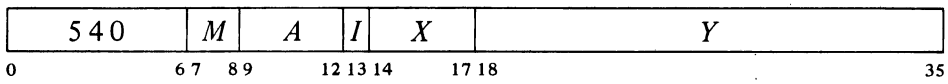
HRLE Half Word Right to Left, Extend



Move the right half of the source word specified by *M* to the left half of the specified destination, and make all bits in the destination right half equal to bit 18 of the source. The source is unaffected, the original contents of the destination are lost.

HRLE	Half Right to Left, Extend	534
HRLEI	Half Right to Left, Extend, Immediate	535
HRLEM	Half Right to Left, Extend, Memory	536
HRLES	Half Right to Left, Extend, Self	537

HRR Half Word Right to Right

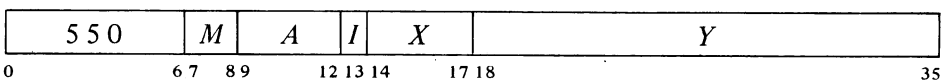


Move the right half of the source word specified by *M* to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

HRR	Half Right to Right	540
HRRI	Half Right to Right Immediate	541
HRRM	Half Right to Right Memory	542
HRRS	Half Right to Right Self	543

If *A* is zero, HRRS is a no-op; otherwise it is equivalent to MOVE.

HRRZ Half Word Right to Right, Zeros



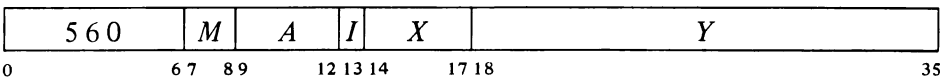
Move the right half of the source word specified by *M* to the right half of the

specified destination, and clear the destination left half. The source is unaffected, the original contents of the destination are lost.

HRRZ	Half Right to Right, Zeros	550
HRRZI	Half Right to Right, Zeros, Immediate	551
HRRZM	Half Right to Right, Zeros, Memory	552
HRRZS	Half Right to Right, Zeros, Self	553

HRRZI loads the word 0,*E* into AC. If *A* is zero, HRRZS merely clears the left half of location *E*.

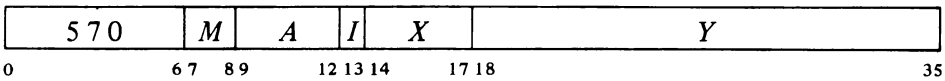
HRRO Half Word Right to Right, Ones



Move the right half of the source word specified by *M* to the right half of the specified destination, and set the destination left half to all 1s. The source is unaffected, the original contents of the destination are lost.

HRRO	Half Right to Right, Ones	560
HRROI	Half Right to Right, Ones, Immediate	561
HRROM	Half Right to Right, Ones, Memory	562
HRROS	Half Right to Right, Ones, Self	563

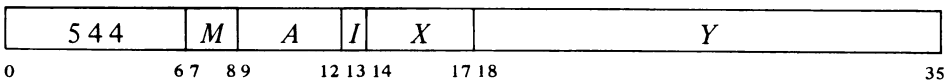
HRRE Half Word Right to Right, Extend



Move the right half of the source word specified by *M* to the right half of the specified destination, and make all bits in the destination left half equal to bit 18 of the source. The source is unaffected, the original contents of the destination are lost.

HRRE	Half Right to Right, Extend	570
HRREI	Half Right to Right, Extend, Immediate	571
HRREM	Half Right to Right, Extend, Memory	572
HRRES	Half Right to Right, Extend, Self	573

HLR Half Word Left to Right



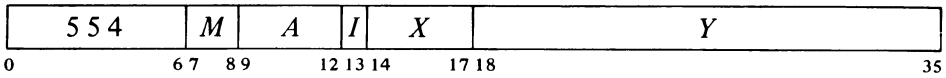
Move the left half of the source word specified by *M* to the right half of the

specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

HLR	Half Left to Right	544
HLRI	Half Left to Right Immediate	545
HLRM	Half Left to Right Memory	546
HLRS	Half Left to Right Self	547

HLRI merely clears AC right.

HLRZ Half Word Left to Right, Zeros

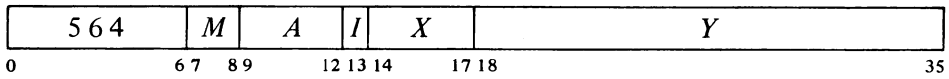


Move the left half of the source word specified by *M* to the right half of the specified destination, and clear the destination left half. The source is unaffected, the original contents of the destination are lost.

HLRZ	Half Left to Right, Zeros	554
HLRZI	Half Left to Right, Zeros, Immediate	555
HLRZM	Half Left to Right, Zeros, Memory	556
HLRZS	Half Left to Right, Zeros, Self	557

HLRZI merely clears AC and is thus equivalent to HLLZI.

HLRO Half Word Left to Right, Ones

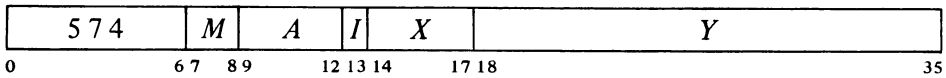


Move the left half of the source word specified by *M* to the right half of the specified destination, and set the destination left half to all 1s. The source is unaffected, the original contents of the destination are lost.

HLRO	Half Left to Right, Ones	564
HLROI	Half Left to Right, Ones, Immediate	565
HLROM	Half Left to Right, Ones, Memory	566
HLROS	Half Left to Right, Ones, Self	567

HLROI sets AC to all 1s in the left half, all 0s in the right.

HLRE Half Word Left to Right, Extend



Move the left half of the source word specified by *M* to the right half of the

specified destination, and make all bits in the destination left half equal to bit 0 of the source. The source is unaffected, the original contents of the destination are lost.

HLRE	Half Left to Right, Extend	574
HLREI	Half Left to Right, Extend, Immediate	575
HLREM	Half Left to Right, Extend, Memory	576
HLRES	Half Left to Right, Extend, Self	577

HLREI is equivalent to HLRZI (it merely clears AC).

EXAMPLES. The half word transmission instructions are very useful for handling addresses, and they provide a convenient means of setting up an accumulator whose right half is to be used for indexing while a control count is kept in the left half. *Eg* this pair of instructions loads the 18-bit numbers *M* and *N* into the left and right halves respectively of an accumulator that is addressed symbolically as XR.

```
HRLZI   XR,M
HRRI    XR,N
```

It is not necessary to clear the other half of XR when loading the first half word. But any instruction that modifies the other half is faster than the corresponding instruction that does not, as the latter must fetch the destination word in order to save half of it. (The difference does not apply to self mode, for here the source and destination are the same.)

Of course the source program must somewhere define the value of the symbol XR as an octal number between 1 and 17.

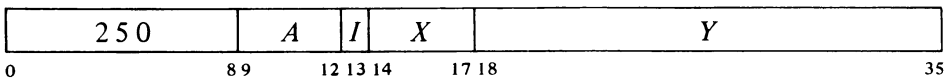
Suppose that at some point we wish to use the two halves of XR independently as operands (taken as 18-bit positive numbers) for computations. We can begin by moving XR left to the right half of another accumulator AC and leaving the contents of XR right alone in XR.

```
HLRZM   XR,AC
HLLI    XR,             ;Clear XR left
```

2.2 FULL WORD DATA TRANSMISSION

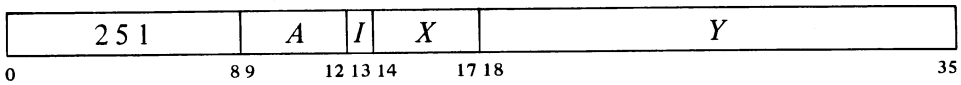
These are the instructions whose basic purpose is to move one or more full words of data from one place to another, usually from an accumulator to a memory location or vice versa. In a few cases instructions may perform minor arithmetic operations, such as forming the negative or the magnitude of the word being processed.

EXCH **Exchange**



Move the contents of location *E* to AC and move AC to location *E*.

BLT Block Transfer



Beginning at the location addressed by AC left, move words to another area of memory beginning at the location addressed by AC right. Continue until a word is moved to location E. The total number of words in the block is thus $E - AC_R + 1$. If $AC_R \geqslant E$, the BLT moves one word to location AC_R.

For a reverse BLT procedure (highest addresses first), refer to the POP instruction on page 2-13. ▲

CAUTION

Priority interrupts are allowed during the execution of this instruction, following the processing of each word. If an interrupt or a page failure occurs, the BLT stores the source and destination addresses for the next word in AC, so when the processor restarts upon the return to the interrupted program, it actually resumes at the correct point within the BLT.

Therefore, unless the interrupt system is inactive and paging is turned off, A and X must not address the same register as this would produce a different effective address upon resumption should an interrupt or page failure occur; and the instruction must not attempt to load an accumulator addressed either by A or X unless it is the final location being loaded. Furthermore, the program cannot assume that AC is the same after the BLT as it was before.

In the KL10, if AC is not in the destination block, the final result in AC is the address of the first word following the source block in AC_L and the address of the first word following the destination block in AC_R.

EXAMPLES. This pair of instructions loads the accumulators from memory locations 2000–2017.

```
HRLZI   17,2000     ;Put 2000 000000 in AC 17
BLT     17,17
```

But to transfer the block in the opposite direction requires that one accumulator first be made available to the BLT:

```
MOVEM   17,2017     ;Move AC 17 to 2017 in memory
MOVEI   17,2000     ;Move the number 2000 to AC 17
BLT     17,2016
```

If at the time the accumulators were loaded the program had placed in location 2017 the control word necessary for storing them back in the same block (2000), the three instructions above could be replaced by

```
EXCH    17,2017
BLT     17,2016
```

Move Instructions

Besides the move instructions for single words there are also

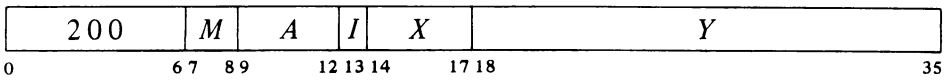
Each of these instructions moves a single word, which may be changed in the process (*eg* its two halves may be swapped). There are four instructions,

each with four modes that determine the source and destination of the word moved.

Mode	*Suffix*	*Source*	*Destination*
Basic		*E*	AC
Immediate	I	The word 0,*E*	AC
Memory	M	AC	*E*
Self	S	*E*	*E*, but also AC if *A* is nonzero

four transmission instructions that handle double length operands (operands of two adjacent words). These are available, however, only in the KI10; and since they are principally for use in hardware double precision floating point operations, they are described with the floating point instructions in §2.6

MOVE **Move**

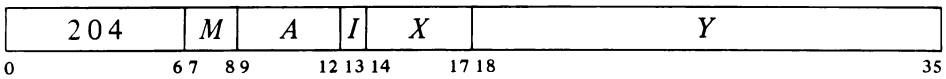


Move one word from the source to the destination specified by *M*. The source is unaffected, the original contents of the destination are lost.

MOVE	Move	200
MOVEI	Move Immediate	201
MOVEM	Move to Memory	202
MOVES	Move to Self	203

MOVEI loads the word 0,*E* into AC and is thus equivalent to HRRZI. If *A* is zero, MOVES is a no-op; otherwise it is equivalent to MOVE.

MOVS **Move Swapped**

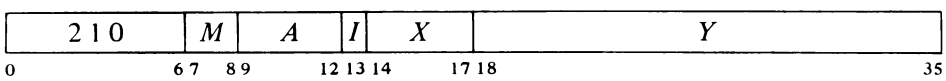


Interchange the left and right halves of the word from the source specified by *M* and move it to the specified destination. The source is unaffected, the original contents of the destination are lost.

MOVS	Move Swapped	204
MOVSI	Move Swapped Immediate	205
MOVSM	Move Swapped to Memory	206
MOVSS	Move Swapped to Self	207

Swapping halves in immediate mode loads the word *E*,0 into AC. MOVSI is thus equivalent to HRLZI.

MOVN **Move Negative**



Negate the word from the source specified by *M* and move it to the specified destination. If the source word is fixed point -2^{35} (400000 000000) set the

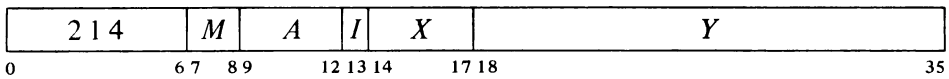
In the KI10 a move executed as an interrupt instruction can set no flags.

Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} sets the flags, but this is not a normalized number.) If the source word is zero, set Carry 0 and Carry 1. The source is unaffected, the original contents of the destination are lost. Setting Overflow also sets the Trap 1 flag in the KI10 and KL10.

MOVN	Move Negative	210
MOVNI	Move Negative Immediate	211
MOVNM	Move Negative to Memory	212
MOVNS	Move Negative to Self	213

MOVNI loads AC with the negative of the word 0,*E* and can set no flags.

MOVM Move Magnitude



Take the magnitude of the word contained in the source specified by *M* and move it to the specified destination. If the source word is fixed point -2^{35} (400000 000000) set the Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} sets the flags, but this is not a normalized number.) The source is unaffected, the original contents of the destination are lost. Setting Overflow also sets the Trap 1 flag in the KI10 and KL10.

In the KI10 a move executed as an interrupt instruction can set no flags.

MOVM	Move Magnitude	214
MOVMI	Move Magnitude Immediate	215
MOVMM	Move Magnitude to Memory	216
MOVMS	Move Magnitude to Self	217

The word 0,*E* is equivalent to its magnitude, so MOVMI is equivalent to MOVEI.

An example at the end of the preceding section demonstrates the use of a pair of immediate-mode half word transfers to load an address and a control count into an accumulator. The same result can be attained by a single move instruction. This saves time but still requires two locations. *Eg* if the number 200 001400 is stored in location M, the instruction

```
MOVE    AC,M
```

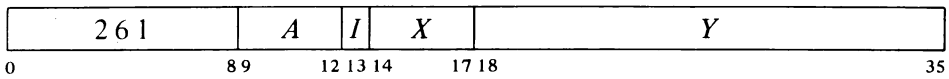
loads 200 into AC left and 1400 into AC right. If the same word, or its negative, or with its halves swapped, must be loaded on several occasions, then both time and space can be saved as each transfer requires only a single move instruction that references M.

Pushdown List

These two instructions insert and remove full words in a pushdown list. The address of the top item in the list is kept in the right half of a pointer in AC,

and the program can keep a control count in the left half. There are also two subroutine-calling instructions that utilize a pushdown list of jump addresses [§2.9].

PUSH Push Down



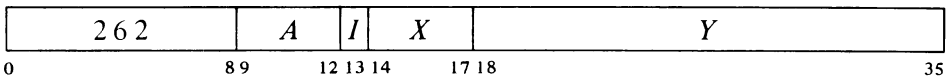
Add one to each half of AC, then move the contents of location E to the location now addressed by AC right. If the addition causes the count in AC left to reach zero, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. The contents of E are unaffected, the original contents of the location added to the list are lost.

In the KI10 a PUSH executed as an interrupt instruction cannot set Trap 2.

Note:

The KA10 increments the two halves of AC by adding 1000001_8 to the entire register. In the KI10 and KL10, the two halves are handled independently.

POP Pop Up



Move the contents of the location addressed by AC right to location E, then subtract one from each half of AC. If the subtraction causes the count in AC left to reach -1, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. The original contents of E are lost.

In the KI10 a POP executed as an interrupt instruction cannot set Trap 2.

In the KA10 and KI10, because of the order in which the operands are stored, the instruction POP AC, AC would load the contents of the location addressed by AC right into AC on top of the pushdown count, destroying it.

In the KL10, POP AC, AC stores the pushdown count after (E), and therefore merely discards the top item from the stack.

Note:

The KA10 decrements the two halves of AC by subtracting 1000001_8 from the entire register. In the KI10 and KL10 the two halves are handled independently.

In the KA10, incrementing and decrementing both halves of AC together is effected by adding and subtracting $1\,000001_8$. Hence a count of -2 in AC left is increased to zero if $2^{18}-1$ is incremented in AC right, and conversely, 1 in AC left is decreased to -1 if zero is decremented in AC right.

A pushdown list is simply a set of consecutive memory locations from which words are read in the order opposite that in which they are written. In more general terms, it is any list in which the only item that can be removed at any given time is the last item in the list. This is usually referred to as "first in, last out" or "last in, first out". Suppose locations *a, b, c, ...* are set aside for a pushdown list. We can deposit data in *a, b, c, d,* then read

A POP can be used to implement a reverse BLT, *ie* to transfer a block of words from one area of memory to another, starting at the largest addresses and proceeding to the smallest. To move a block of N words from a source area to a destination area whose maximum addresses are S and D respectively, the program must first set up a pushdown pointer in accumulator T, where T left contains $N - 1 + 400000$ and T right contains S. The transfer is then effected by this pair of instructions

```
POP     T,D-S(T)
JUMPL   T,.-1
```

where the jump returns to the POP until T left is less than 400000, *ie* until it looks positive. The 400000 added into T left prevents pushdown overflow, but also limits the block to 2^{17} words.

d, then write in *d* and *e*, then read *e*, *d*, *c*, etc.

Note that by trapping or checking overflow and keeping a control count in AC left, the programmer can set a limit to the size of the list by starting the count negative, or he can prevent the program from extracting more words than there are in the list by starting the count at zero, but he cannot do both at once. The common practice is to limit the size of the list.

Pushdown storage is very convenient for a program that can use data stored in this manner as the pointer is initialized only once and only one accumulator is required for the most complex pushdown operations. To initialize a pointer P for a list to be kept in a block of memory beginning at BLIST and to contain at most *N* items, the following suffices.

```
MOVSI   P,-N
HRRI    P,BLIST-1
```

Of course the programmer must define BLIST elsewhere and set aside locations BLIST to BLIST + *N* − 1. Using MACRO to full advantage one could instead give

```
MOVE    P,[IOWD N,BLIST]
```

where the pseudoinstruction

```
IOWD J,K
```

is replaced by a word containing −*J* in the left half and *K* − 1 in the right. Elsewhere there would appear

```
BLIST:  BLOCK N
```

which defines BLIST as the current contents of the location counter and sets aside the *N* locations beginning at that point.

In the PDP-10 the pushdown list is kept in a random access core memory, so the restrictions on order of entry and removal of items actually apply only to the standard addressing by the pointer in pushdown instructions – other addressing methods can reference any item at any time. The most convenient way to do this is to use the right half of the pointer as an index register. To move the last entry to accumulator AC we need simply give

```
MOVE    AC,(P)
```

Of course this does not shorten the list – the word moved remains the last item in it.

One usually regards an index register as supplying an additive factor for a basic address contained in an instruction word, but the index register can supply the basic address and the instruction the additive factor. Thus we can retrieve the next to last item by giving

```
MOVE    AC,-1(P)
```

and so forth. Similarly

```
PUSH    P,-3(P)
```

adds the third to last item to the end of the list:

```
POP     P,-2(P)
```

removes the last item and inserts it in place of the next to last item in the shortened list.

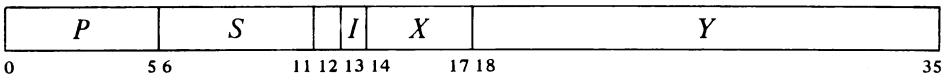
Note that *E* is calculated before the contents of P are changed.

2.3 BYTE MANIPULATION

In a KA10 without byte manipulation hardware, all of the instructions presented in this section are trapped as unassigned codes [§2.10].

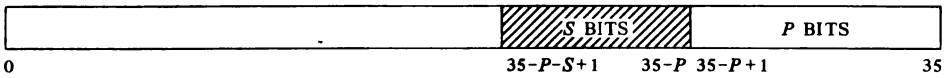
This set of five instructions allows the programmer to pack or unpack bytes of any length anywhere within a word. Movement of a byte is always between AC and a memory location: a deposit instruction takes a byte from the right end of AC and inserts it at any desired position in the memory location; a load instruction takes a byte from any position in the memory location and places it right-justified in AC.

The byte manipulation instructions have the standard memory reference format, but the effective address *E* is used to retrieve a pointer, which is used in turn to locate the byte or the place that will receive it. The pointer has the format



Bit 12 is reserved for future use and should be 0.

where *S* is the size of the byte as a number of bits, and *P* is its position as the number of bits remaining at the right of the byte in the word (*eg* if *P* is 3 the rightmost bit of the byte is bit 32 of the word). The rest of the pointer is interpreted in the same way as in an instruction: *I*, *X* and *Y* are used to calculate the address of the location that is the source or destination of the byte. Thus the pointer aims at a word whose format is



where the shaded area is the byte.

To facilitate processing a series of bytes, several of the byte instructions increment the pointer, *ie* modify it so that it points to the next byte position in a set of memory locations. In the KL10, one of these instructions may modify the pointer so that it points to any byte. Bytes are processed from left to right in a word, so incrementing merely replaces the current value of P by P – S, unless there is insufficient space in the present location for another byte of the specified size ($P - S < 0$). In this case *Y* is increased by one to point to the next consecutive location, and *P* is set to $36 - S$ to point to the first byte at the left in the new location.

CAUTION (KA10 ONLY)

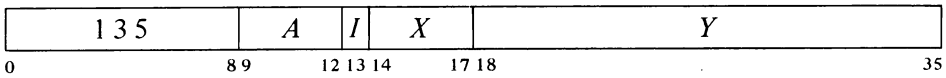
Do not allow *Y* to reach maximum value. The whole pointer is incre-

In the KL10 and KI10, incrementing maximum Y produces a zero address without affecting X.

mented, so if Y is $2^{18}-1$ it becomes zero and X is also incremented. The address calculation for the pointer uses the original X, but if a priority interrupt should occur before the calculation is complete, the incremented X is used when the instruction is repeated.

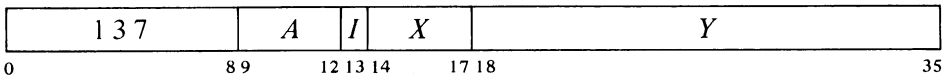
Among these five instructions one simply increments the pointer, the others load or deposit a byte with or without incrementing.

LDB Load Byte



Retrieve a byte of S bits from the location and position specified by the pointer contained in location E, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected, the original contents of AC are lost.

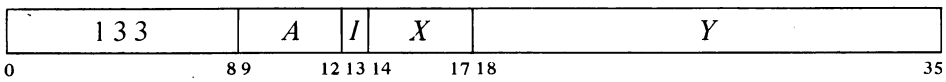
DPB Deposit Byte



Deposit the right S bits of AC into the location and position specified by the pointer contained in location E. The original contents of the bits that receive the byte are lost, AC and the remaining bits of the deposit location are unaffected.

IBP Increment Byte Pointer

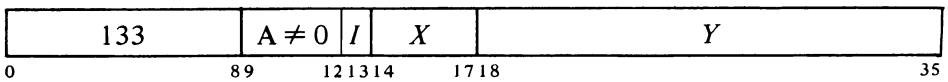
The A field must be zero. A nonzero A field is the ADJBP instruction below.



Increment the byte pointer in location E as explained above.

ADJBP Adjust Byte Pointer (KL10 only)

The A field must be nonzero. A zero A field is the IBP instruction.



Retrieve the byte pointer from location E. Adjust the pointer by the number of bytes specified by AC. Place the adjusted byte pointer in AC. The location containing the original byte pointer is unaffected; the original contents of AC are lost.

If AC contains a positive value, ADJBP advances the pointer. If AC contains a negative value, ADJBP backs up the pointer. In both cases, the byte alignment is preserved across word boundaries.

The term *byte alignment* refers to the position of the left-most byte of a word, as implied by the P and S fields. Numerically, it is the remainder of

$$\frac{36\text{-}P}{S}$$

Ordinary strings are packed with the alignment equal to zero because IBP, ILDP, and IDPB force the alignment to zero at every word boundary. ADJBP, however, preserves the byte alignment across word boundaries.

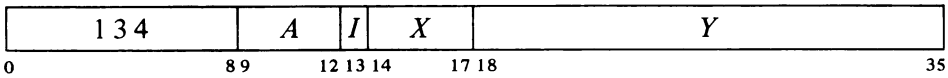
ADJBP always returns a byte pointer designating a complete byte within a word. For example, ADJBP by 0 bytes on a byte pointer with P equal to 36 will return a byte pointer addressing the right-most byte in the previous word.

Adjustment is performed by dividing (AC) by the number of bytes per word, which is computed as:

$$\text{IFIX}\left(\frac{36\text{-}P}{S}\right) + \text{IFIX}\left(\frac{P}{S}\right)$$

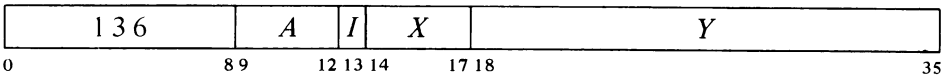
If the number of bytes that fit in a word is 0 (*eg* $S > 36$), then ADJBP will set No Divide and go to the next instruction without modifying AC or memory.

ILDB Increment Pointer and Load Byte



Increment the byte pointer in location *E* as explained above. Then retrieve a byte of *S* bits from the location and position specified by the newly incremented pointer, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected, the original contents of AC are lost.

IDPB Increment Pointer and Deposit Byte



Increment the byte pointer in location E as explained above. Then deposit the right S bits of AC into the location and position specified by the newly incremented pointer. The original contents of the bits that receive the byte are lost, AC and the remaining bits of the deposit location are unaffected.

Note that in the pair of instructions that both increment the pointer and process a byte, it is the *modified* pointer that determines the byte location and position. Hence to unpack bytes from a block of memory, the program should set up the pointer to point to a byte just *before* the first desired, and then load them with a loop containing an ILDB. If the first byte is at the left end of a word, this is most easily done by initializing the pointer with a P of 36 (44_8). Incrementing then replaces the 36 with $36-S$ to point to the first byte. At any time that the program might inspect the pointer during execution of a series of ILDBs or IDPBs, it points to the last byte processed (this may not be true when the pointer is tested from an interrupt routine [§2.13]).

Special Considerations. If S is greater than P and also greater than 36, incrementing produces a new P equal to $100-S$ rather than $36-S$. For $S>36$ the byte is at most the entire word; for $P\geqslant 36$ no byte is processed (loading merely clears AC). If both P and S are less than 36 but $P+S>36$, a byte of size $36-P$ is loaded from position P, or the right $36-P$ bits of the byte are deposited in position P.

The Extended Instruction Set executed by the KL10 is described in a separate supplement.

2.4 LOGIC

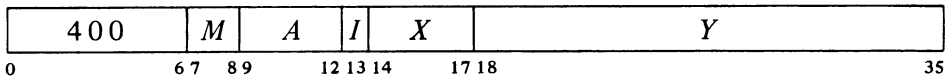
For logical operations the PDP-10 has instructions for shifting and rotating as well as for performing the complete set of sixteen Boolean functions of two variables (including those in which the result depends on only one or neither variable). The Boolean functions operate bitwise on full words, so each instruction actually performs thirty-six logical operations simultaneously. Thus in the AND function of two words, each bit of the result is the AND of the corresponding bits of the operands. The table on page 2-23 lists the bit configurations that result from the various operand configurations for all instructions.

Each Boolean instruction has four modes that determine the source of the non-AC operand, if any, and the destination of the result. For an instruction without an operand (one that merely clears a location or sets it to all 1s) the modes differ only in the destination of the result, so basic and immediate

modes are equivalent. The same is true also of an instruction that uses only an AC operand. When specified by the mode, the result goes to the accumulator addressed by *A*, even when there is no AC operand.

Mode	*Suffix*	*Source of non-AC operand*	*Destination of result*
Basic		*E*	AC
Immediate	I	The word 0, *E*	AC
Memory	M	*E*	*E*
Both	B	*E*	AC and *E*

SETZ **Set to Zeros**



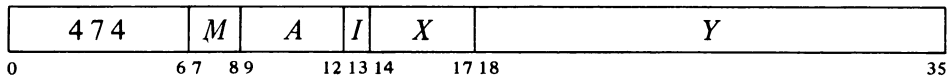
Change the contents of the destination specified by *M* to all 0s.

SETZ and SETZI are equivalent (both merely clear AC). In them, *I*, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored).

MACRO also recognizes CLEAR, CLEARI, CLEARM and CLEARB as equivalent to the set-to-zeros mnemonics.

SETZ	Set to Zeros	400
SETZI	Set to Zeros Immediate	401
SETZM	Set to Zeros Memory	402
SETZB	Set to Zeros Both	403

SETO **Set to Ones**

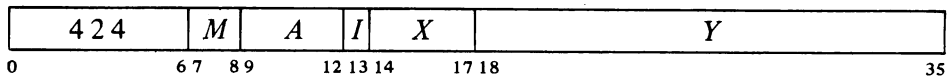


Change the contents of the destination specified by *M* to all 1s.

SETO and SETOI are equivalent. In them, *I*, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored).

SETO	Set to Ones	474
SETOI	Set to Ones Immediate	475
SETOM	Set to Ones Memory	476
SETOB	Set to Ones Both	477

SETA **Set to AC**



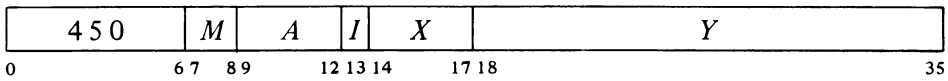
Make the contents of the destination specified by *M* equal to AC.

SETA	Set to AC	424
SETAI	Set to AC Immediate	425
SETAM	Set to AC Memory	426
SETAB	Set to AC Both	427

SETA and SETAI are no-ops. In them, *I*, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored).

SETAM and SETAB are both equivalent to MOVEM (all move AC to location *E*).

SETCA Set to Complement of AC

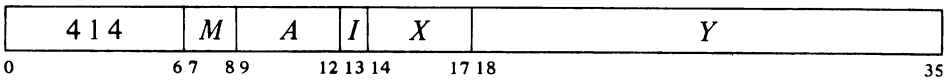


Change the contents of the destination specified by *M* to the complement of AC.

SETCA	Set to Complement of AC	450
SETCAI	Set to Complement of AC Immediate	451
SETCAM	Set to Complement of AC Memory	452
SETCAB	Set to Complement of AC Both	453

SETCA and SETCAI are equivalent (both complement AC). In them, *I*, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored).

SETM Set to Memory

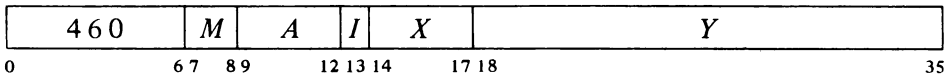


Make the contents of the destination specified by *M* equal to the specified operand.

SETM	Set to Memory	414
SETMI	Set to Memory Immediate	415
SETMM	Set to Memory Memory	416
SETMB	Set to Memory Both	417

SETM and SETMB are equivalent to MOVE. SETMI moves the word 0,*E* to AC and is thus equivalent to MOVEI. SETMM is a no-op that references memory.

SETCM Set to Complement of Memory

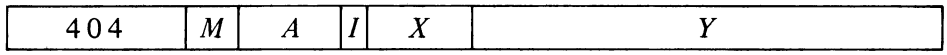


Change the contents of the destination specified by *M* to the complement of the specified operand.

SETCM	Set to Complement of Memory	460
SETCMI	Set to Complement of Memory Immediate	461
SETCMM	Set to Complement of Memory Memory	462
SETCMB	Set to Complement of Memory Both	463

SETCMI moves the complement of the word 0,*E* to AC. SETCMM complements location *E*.

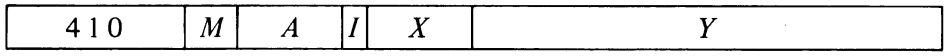
AND And with AC



Change the contents of the destination specified by *M* to the AND function of the specified operand and AC.

AND	And	404
ANDI	And Immediate	405
ANDM	And to Memory	406
ANDB	And to Both	407

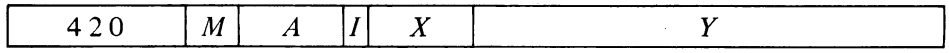
ANDCA And with Complement of AC



Change the contents of the destination specified by *M* to the AND function of the specified operand and the complement of AC.

ANDCA	And with Complement of AC	410
ANDCAI	And with Complement of AC Immediate	411
ANDCAM	And with Complement of AC to Memory	412
ANDCAB	And with Complement of AC to Both	413

ANDCM And Complement of Memory with AC

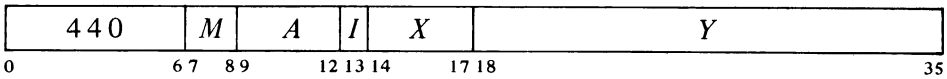


Change the contents of the destination specified by *M* to the AND function of the complement of the specified operand and AC.

ANDCM	And Complement of Memory	420
ANDCMI	And Complement of Memory Immediate	421

ANDCMM	And Complement of Memory to Memory	422
ANDCMB	And Complement of Memory to Both	423

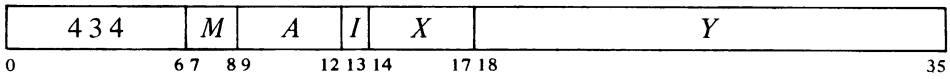
ANDCB And Complements of Both



Change the contents of the destination specified by *M* to the AND function of the complements of both the specified operand and AC. The result is the NOR function of the operands.

ANDCB	And Complements of Both	440
ANDCBI	And Complements of Both Immediate	441
ANDCBM	And Complements of Both to Memory	442
ANDCBB	And Complements of Both to Both	443

IOR Inclusive Or with AC

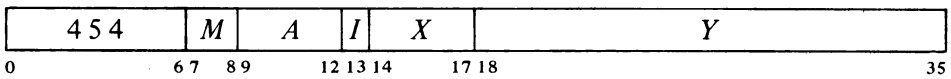


Change the contents of the destination specified by *M* to the inclusive OR function of the specified operand and AC.

IOR	Inclusive Or	434
IORI	Inclusive Or Immediate	435
IORM	Inclusive Or to Memory	436
IORB	Inclusive Or to Both	437

MACRO also recognizes OR, ORI, ORM and ORB as equivalent to the inclusive OR mnemonics.

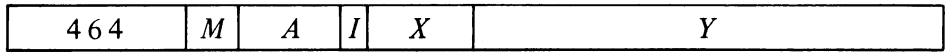
ORCA Inclusive Or with Complement of AC



Change the contents of the destination specified by *M* to the inclusive OR function of the specified operand and the complement of AC.

ORCA	Or with Complement of AC	454
ORCAI	Or with Complement of AC Immediate	455
ORCAM	Or with Complement of AC to Memory	456
ORCAB	Or with Complement of AC to Both	457

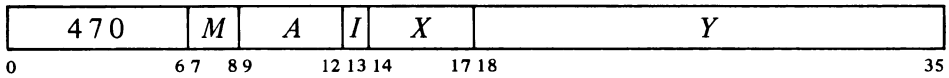
ORCM Inclusive Or Complement of Memory with AC



Change the contents of the destination specified by *M* to the inclusive OR function of the complement of the specified operand and AC.

ORCM	Or Complement of Memory	464
ORCMI	Or Complement of Memory Immediate	465
ORCMM	Or Complement of Memory to Memory	466
ORCMB	Or Complement of Memory to Both	467

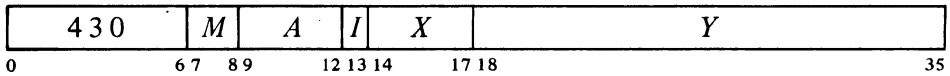
ORCB Inclusive Or Complements of Both



Change the contents of the destination specified by *M* to the inclusive OR function of the complements of both the specified operand and AC. The result is the NAND function of the operands.

ORCB	Or Complements of Both	470
ORCBI	Or Complements of Both Immediate	471
ORCBM	Or Complements of Both to Memory	472
ORCBB	Or Complements of Both to Both	473

XOR Exclusive Or with AC

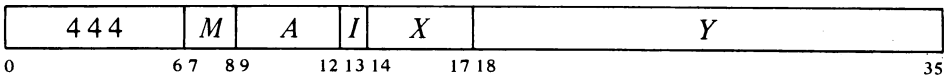


Change the contents of the destination specified by *M* to the exclusive OR function of the specified operand and AC.

XOR	Exclusive Or	430
XORI	Exclusive Or Immediate	431
XORM	Exclusive Or to Memory	432
XORB	Exclusive Or to Both	433

The original contents of the destination can be recovered except in XORB, where both operands are replaced by the result. In the other three modes the replaced operand is restored by repeating the instruction in the same mode, *ie* by taking the exclusive OR of the remaining operand and the result.

EQV **Equivalence with AC**



Change the contents of the destination specified by *M* to the complement of the exclusive OR function of the specified operand and AC (the result has 1s wherever the corresponding bits of the operands are the same).

EQV	Equivalence	444
EQVI	Equivalence Immediate	445
EQVM	Equivalence to Memory	446
EQVB	Equivalence to Both	447

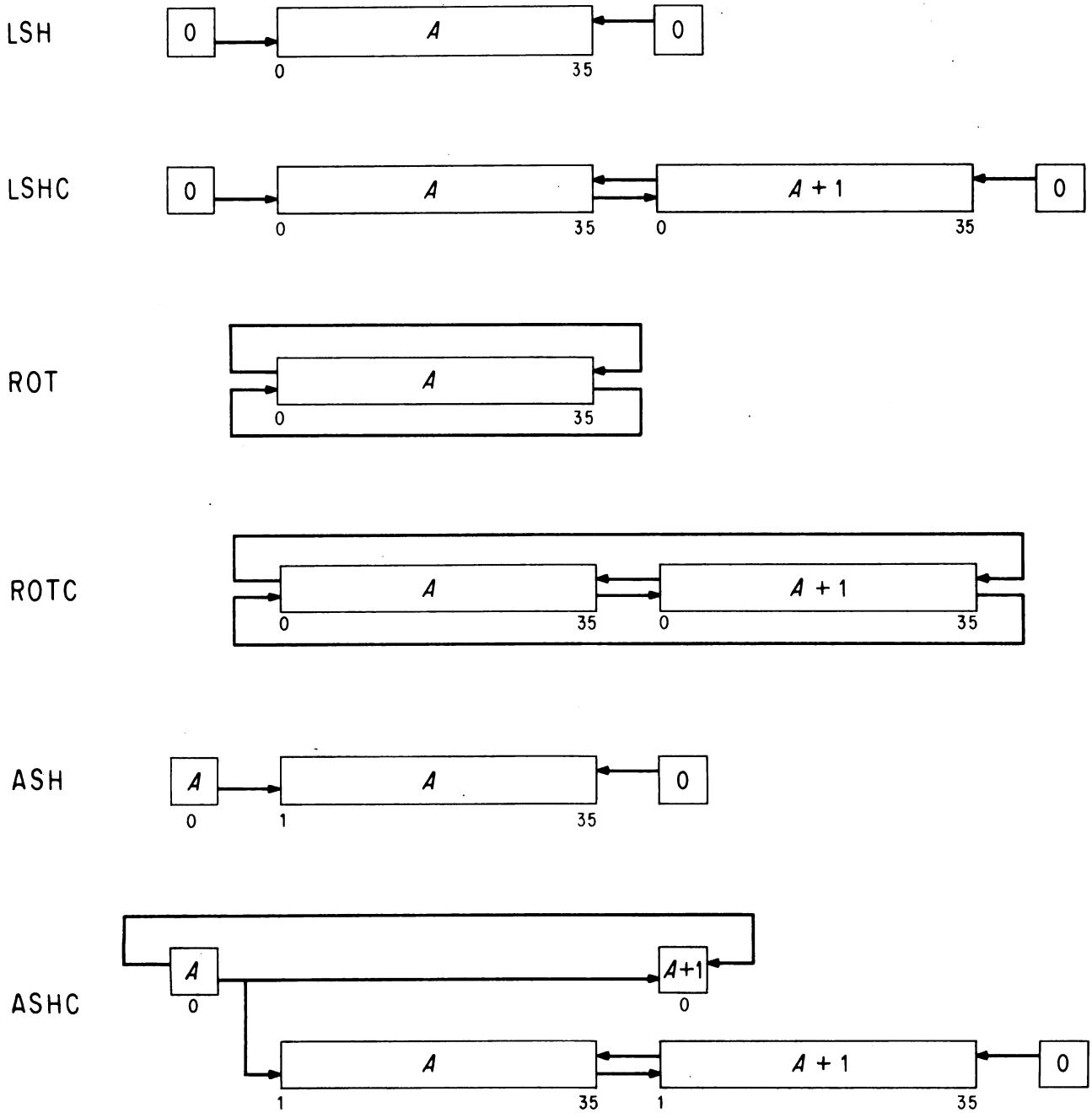
The original contents of the destination can be recovered except in EQVB, where both operands are replaced by the result. In the other three modes the replaced operand is restored by repeating the instruction in the same mode, *ie* by taking the equivalence function of the remaining operand and the result.

For the four possible bit configurations of the two operands, the above sixteen instructions produce the following results. In each case the result as listed is equal to bits 3–6 of the instruction word.

AC	0	1	0	1
Mode Specified Operand	0	0	1	1
SETZ	0	0	0	0
AND	0	0	0	1
ANDCA	0	0	1	0
SETM	0	0	1	1
ANDCM	0	1	0	0
SETA	0	1	0	1
XOR	0	1	1	0
IOR	0	1	1	1
ANDCB	1	0	0	0
EQV	1	0	0	1
SETCA	1	0	1	0
ORCA	1	0	1	1
SETCM	1	1	0	0
ORCM	1	1	0	1
ORCB	1	1	1	0
SETO	1	1	1	1

Shift and Rotate

The remaining logical instructions shift or rotate right or left the contents of AC or the contents of two accumulators, A and $A+1$ (mod 20_8), concatenated into a 72-bit register with A on the left. The illustration below shows the movement of information these instructions produce in the accu-

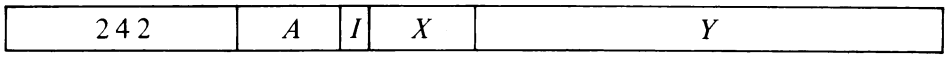


ACCUMULATOR BIT FLOW IN SHIFT AND ROTATE INSTRUCTIONS

mulators. In a (logical) shift the contents of a register are moved bit-to-bit with 0s brought in at the end being vacated; information shifted out at the other end is lost. [*For a discussion of arithmetic shifting see* §2.5.] In rotation the contents are moved cyclically such that information rotated out at one end is put in at the other.

The number of places moved is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective shift E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the shift directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the shift. A positive E produces motion to the left, a negative E to the right. In the KA10, maximum movement is 255 places. The KI10 eliminates redundant movement by logical shifting at most 72 places regardless of the ▲ value of E, and rotating E mod 72 places (except 72 places if E is a nonzero multiple of 72).

LSH Logical Shift

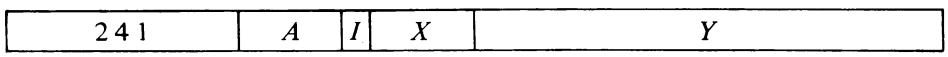


Shift AC the number of places specified by E. If E is positive, shift left bringing 0s into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right bringing 0s into bit 0; data shifted out of bit 35 is lost.

LSHC Logical Shift Combined

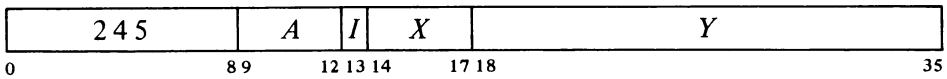
Concatenate accumulators A and $A+1$ with A on the left, and shift the 72-bit combination the number of places specified by E. If E is positive, shift left bringing 0s into bit 71 (bit 35 of AC $A+1$); bit 36 is shifted into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right bringing 0s into bit 0; bit 35 is shifted into bit 36; data shifted out of bit 71 is lost.

ROT Rotate



Rotate AC the number of places specified by E. If E is positive, rotate left; bit 0 is rotated into bit 35. If E is negative, rotate right; bit 35 is rotated into bit 0.

ROTC Rotate Combined



Concatenate accumulators A and $A+1$ with A on the left, and rotate the 72-bit combination the number of places specified by E. If E is positive, rotate left; bit 0 is rotated into bit 71 (bit 35 of AC $A+1$) and bit 36 into bit 35. If E is negative, rotate right; bit 35 is rotated into bit 36 and bit 71 into bit 0.

2.5 FIXED POINT ARITHMETIC

For fixed point arithmetic the PDP-10 has instructions for arithmetic shifting (which is essentially multiplication by a power of 2) as well as for performing addition, subtraction, multiplication and division of numbers in fixed point format [§1.1]. In such numbers the position of the binary point is arbitrary (the programmer may adopt any point convention). The add and subtract instructions involve single or (KL10 only) double length numbers, whereas multiply supplies a double or (KL10 only) quadruple length product, and divide uses a double or (KL10 only) quadruple length dividend. The high and low order words respectively of a double length fixed point number are in accumulators A and A+1 (mod 20_8), where the magnitude is the 70-bit string in bits 1-35 of the two words and the signs of the two are identical. The four words respectively of a quadruple fixed pointer number are in accumulators A, A+1, A+2, and A+3 (mod 20_8), where the magnitude is the 140-bit string in bits 1-35 of the four words, and the signs of the four are the same. There are also integer multiply and divide instructions that involve only single length numbers and are especially suited for handling smaller integers, particularly those of eighteen bits or less such as addresses (of course they can be used for small fractions as well provided the programmer keeps track of the binary point). For convenience in the following, all operands are assumed to be integers (binary point at the right).

The processor has four flags, Overflow, Carry 0, Carry 1 and No Divide, that indicate when the magnitude of a number is or would be larger than can be accommodated. Carry 0 and Carry 1 actually detect carries out of bits 0 and 1 in certain instructions that employ fixed point arithmetic operations: the add and subtract instructions treated here, the move instructions that produce the negative or magnitude of the word moved [§2.2], and the arithmetic test instructions that increment or decrement the test word [§2.7]. In these instructions an incorrect result is indicated – and the Overflow flag set – if the carries are different, *ie* if there is a carry into the sign but not out of it, or vice versa. The Overflow flag is also set by No Divide being set, which means the processor has failed to perform a division because the magnitude of the dividend is greater than or equal to that of the divisor, or in integer divide, simply that the divisor is zero. In other overflow cases

Overflow is determined directly from the carries, not from the carry flags, as their states may reflect events in previous instructions.

only Overflow itself is set: these include too large a product in multiplication, too large a number to convert to fixed point [§2.6], and loss of significant bits in left arithmetic shifting. In the KI10 any condition that sets Overflow also sets the Trap 1 flag.

In the KI10 an arithmetic instruction executed as an interrupt instruction can set no flags.

These flags can be read and controlled by certain program control instructions [§§2.9, 2.10]. KI10 overflow is handled by trapping through the

setting of Trap 1 [§2.9], but in the KA10, the program must make direct use of the Overflow flag, which is available as a processor condition (via an in-out instruction) that can request a priority interrupt if enabled [§2.14]. The conditions detected can only set the arithmetic flags and the hardware does not clear them, so the program must clear them before an instruction if they are to give meaningful information about the instruction afterward. However, the program can check the flags following a series of instructions to determine whether the entire series was free of the types of error detected.

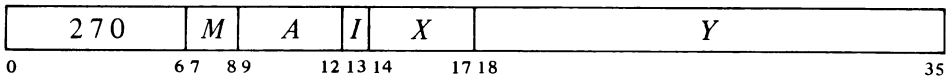
User overflow is handled by the Monitor according to instructions from the user. Refer to Chapter 3 of *DECsystem-10 Monitor Calls.*

Besides indicating error types, the carry flags facilitate performing multiple precision arithmetic.

All but the shift instructions have four modes that determine the source of the non-AC operand and the destination of the result.

Mode	*Suffix*	*Source of non-AC operand*	*Destination of result*
Basic		*E*	AC
Immediate	I	The word 0, *E*	AC
Memory	M	*E*	*E*
Both	B	*E*	AC and *E*

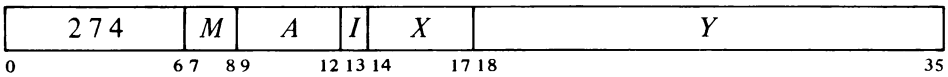
ADD Add



Add the operand specified by *M* to AC and place the result in the specified destination. If the sum is $\geq 2^{35}$ set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the sum less 2^{35}. If the sum is $< -2^{35}$ set Overflow and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the sum plus 2^{35}. Set both carry flags if both summands are negative, or their signs differ and their magnitudes are equal or the positive one is the greater in magnitude.

ADD	Add	270
ADDI	Add Immediate	271
ADDM	Add to Memory	272
ADDB	Add to Both	273

SUB Subtract

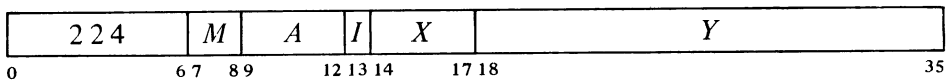


Subtract the operand specified by *M* from AC and place the result in the specified destination. If the difference is $\geq 2^{35}$ set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{35}. If the difference is $< -2^{35}$ set Overflow and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to

the difference plus 2^{35}. Set both carry flags if the signs of the operands are the same and AC is the greater or the two are equal, or the signs of the operands differ and AC is negative.

SUB	Subtract	274
SUBI	Subtract Immediate	275
SUBM	Subtract to Memory	276
SUBB	Subtract to Both	277

MUL Multiply



Multiply AC by the operand specified by M, and place the high order word of the double length result in the specified destination. If M specifies AC as a destination, place the low order word in accumulator $A+1$. If both operands are -2^{35} set Overflow; the double length result stored is -2^{70}.

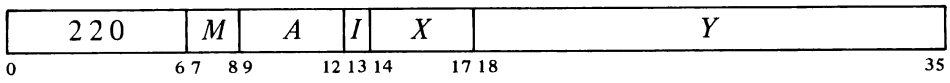
▲ Remember that bit 0 of the low order word is equal to the sign of the product.

CAUTION

In the KA10, an AC operand of -2^{35} is treated as though it were $+2^{35}$, producing the incorrect sign in the product.

MUL	Multiply	224
MULI	Multiply Immediate	225
MULM	Multiply to Memory	226
MULB	Multiply to Both	227

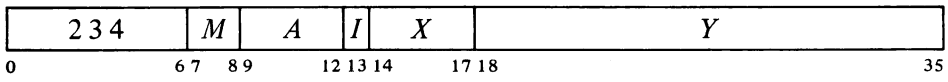
IMUL Integer Multiply



Multiply AC by the operand specified by M, and place the sign and the 35 low order magnitude bits of the product in the specified destination. Set Overflow if the product is $\geqslant 2^{35}$ or $< -2^{35}$ (*ie* if the high order word of the double length product is not null); the high order word is lost.

IMUL	Integer Multiply	220
IMULI	Integer Multiply Immediate	221
IMULM	Integer Multiply to Memory	222
IMULB	Integer Multiply to Both	223

DIV Divide



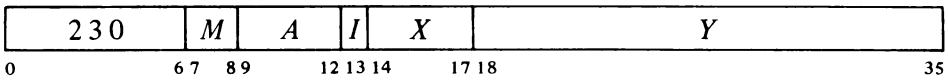
▲ If the high order word of the magnitude of the double length number in

accumulators A and $A+1$ is greater than or equal to the magnitude of the operand specified by M, set Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way. Otherwise divide the double length number contained in accumulators A and $A+1$ by the specified operand, calculating a quotient of 35 magnitude bits including leading zeros. Place the unrounded quotient in the specified destination. If M specifies AC as a destination, place the remainder, with the same sign as the dividend, in accumulator $A+1$.

This restriction is required since the quotient developed would exceed 36 bits.

DIV	Divide	234
DIVI	Divide Immediate	235
DIVM	Divide to Memory	236
DIVB	Divide to Both	237

IDIV **Integer Divide**



If the operand specified by M is zero, or AC contains -2^{35} and the operand specified by M is ± 1, set Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way. Otherwise divide AC by the specified operand, calculating a quotient of 35 magnitude bits including leading zeros. Place the unrounded quotient in the specified destination. If M specifies AC as the destination, place the remainder, with the same sign as the dividend, in accumulator $A+1$. ▲

IDIV	Integer Divide	230
IDIVI	Integer Divide Immediate	231
IDIVM	Integer Divide to Memory	232
IDIVB	Integer Divide to Both	233

EXAMPLES. The integer multiply and divide instructions are very useful for computations on addresses or character codes, or performing any integral operations in which the result is small enough to be accommodated in a single register.

Suppose we wish to reverse the order of the digits in the 6-bit character *abcdef*, where the letters represent the bits of the character. We first duplicate it three times to the left and shift the result left one place producing

*a bcd efa bcd efa bcd efa bcd ef*0

where the bits are grouped corresponding to the octal digits in the word. Anding this with

1 000 100 100 010 010 000 001 000

gives

$$a\ 000\ e00\ b00\ 0f0\ 0c0\ 000\ 00d\ 000$$

Now it just so happens this number is configured such that the residues of the values of its bits modulo $2^8 - 1$ are in exactly the opposite order from the bits of the original character and have the binary orders of magnitude 0–5. In other words this number is equal to the sum of the numbers in the upper row below, and dividing each of these summands by 255 gives the remainder listed in the lower row.

Dividend	$f \times 2^{13}$	$e \times 2^{20}$	$d \times 2^{3}$	$c \times 2^{10}$	$b \times 2^{17}$	$a \times 2^{24}$
Remainder	$f \times 2^{5}$	$e \times 2^{4}$	$d \times 2^{3}$	$c \times 2^{2}$	$b \times 2^{1}$	$a \times 2^{0}$

The remainder in a division is equal to the sum, modulo the divisor, of the remainders left by dividing each of the dividend summands by the same divisor. And the sum of the terms in the lower row is obviously *fedcba*. The above procedure is implemented by this sequence (due to Schroeppel*) where the character is right-justified in accumulator A, and its reverse appears right-justified in accumulator A+1.

*HAKMEM 140, page 78 (*Artificial Intelligence Memorandum, No. 239*, February 29, 1972, MIT Artificial Intelligence Laboratory).

```
IMUL    A,[2020202]         ;4 copies shifted left one
AND     A,[104422010]       ;Pick bits for reverse
IDIVI   A,3777              ;Divide by 2^8 - 1
```

These examples require that the rest of A, outside the character, be clear.

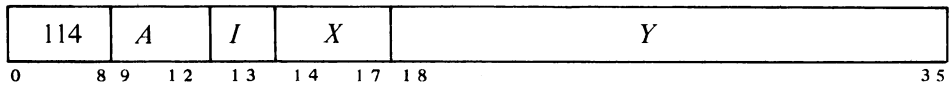
To reverse eight bits we can use a similar procedure (also due to Schroeppel) where again the original character is right-justified in A and its reverse appears right-justified in A+1. But this time we cannot manage the manipulation within a single length word, so we must use multiply, divide, and a pair of ANDs.

```
MUL     A,[100200401002]    ;5 copies in A and A+1
AND     A+1,[20420420020]   ;Pick bits for reverse via
ANDI    A,41                ;residues mod 2^10 - 1
DIVI    A,1777              ;Divide by 2^10 - 1
```

Double Precision Integer Instructions

(KL10 Only)

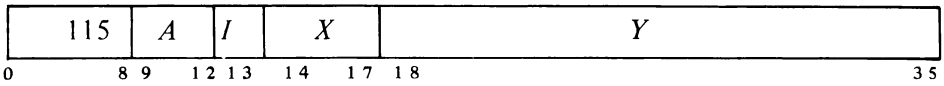
DADD **Double Add**



Double add the contents of *E* and *E*+1 to the contents of AC and AC+1 and place the double word result in AC and AC+1. If the sum is $\geq 2^{70}$, set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the sum less 2^{70}. If the sum is $< -2^{70}$, set Overflow

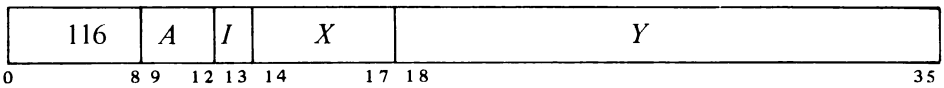
and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the sum plus 2^{70}. Set both Carry 0 and Carry 1 flags if both summands are negative or their signs differ and their magnitudes are equal or the positive one is the greater in magnitude.

DSUB **Double Subtract**



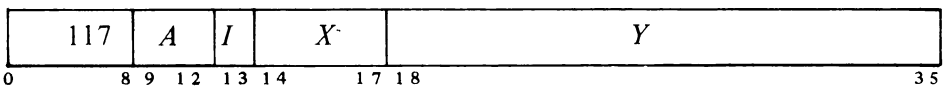
Double subtract the contents of E and E+1 from AC and AC+1 and place the double word result in AC and AC+1. If the difference is $\geqslant 2^{70}$, set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{70}. If the difference is $< -2^{70}$ set Overflow and Carry 0; the result has a plus sign but a magnitude in negative form equal to the difference plus 2^{70}. Set both carry flags if the signs of the operands are the same and the operand in AC and AC+1 is greater or the signs differ and AC, AC+1 is negative.

DMUL **Double Multiply**



Double multiply AC and AC+1 by E and E+1, placing the high order word of the four word result in AC, the next order word in AC+1, the third word in AC+2, and the low order word in AC+3. The signs of the words in E+1 and AC+1 are ignored during multiplication. The signed product is stored in the four ACs where bit 0 of each AC contains the sign of the high order word. If both double word operands before multiplication are -2^{70}, set Overflow; the quadruple length result stored is $-2^{\cdot 40}$.

DDIV **Double Divide**



If the high order double word of the magnitude of the quadruple length number in accumulators A through A+3 is greater than or equal to the magnitude of the double word operand at E, set overflow and no divide, and go immediately to the next instruction without affecting the original ACs or memory operands in any way. Otherwise, divide the quadruple length number contained in accumulators A through A+3 calculating a quotient of 70 magnitude bits including leading zeros. Place the double word quotient in accumulators A and A+1, and the double word remainder, with the same sign as the dividend in A and A+1. The double word operand at E remains unchanged, the original contents of accumulators A through A+3 are lost. Bit 0 of all but the high order word of each operand is ignored in the computation.

Bit 0 of accumulators A and A+1 is set to the sign of the quotient, which is determined algebraically from the signs of the original dividend and divisor. Bit 0 of accumulators *A*+2 and *A*+3 is set to the sign of the remainder which is the same as that of the dividend (unless the remainder is zero).

Arithmetic Shifting

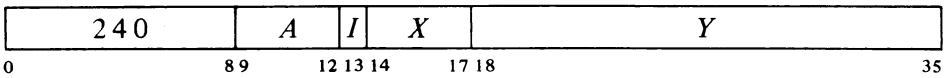
These two instructions produce an arithmetic shift right or left of the number in AC or the double length number in accumulators *A* and *A*+1. Shifting is the movement of the contents of a register bit-to-bit. The operation discussed here is similar to logical shifting [*see* §2.4 *and the illustration on page 2-24*], but in an arithmetic shift only the magnitude part is shifted – the sign is unaffected. In a double length number the 70-bit string made up of the magnitude parts of the two words is shifted, but the sign of the low order word is made equal to the sign of the high order word.

Null bits are brought in at the end being vacated: a left shift brings in 0s at the right, whereas a right shift brings in the equivalent of the sign bit at the left. In either case, information shifted out at the other end is lost. A single

shift left is equivalent to multiplying the number by 2 (provided no bit of significance is shifted out); a shift right divides the number by 2.

The number of places shifted is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective shift E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the shift directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the shift. A positive E produces motion to the left, a negative E to the right; E is thus the power of 2 by which the number is multiplied. In the KA10, maximum movement is 255 places. The KI10 eliminates redundant movement by shifting at most 72 places regardless of the value of E. ▲

ASH Arithmetic Shift



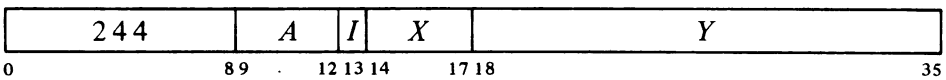
Shift AC arithmetically the number of places specified by E. Do not shift bit 0. If E is positive, shift left bringing 0s into bit 35; data shifted out of bit 1 is lost; set Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative one). If E is negative, shift right bringing 0s into bit 1 if AC is positive, 1s if negative; data shifted out of bit 35 is lost.

An arithmetic right shift truncates a negative result differently from IDIV if 1s are shifted out. The result of the shift is more negative by one than the quotient of IDIV.

To obtain the same quotient that IDIV would give with a dividend in A divided by $N = 2^K$, use

```
SKIPGE  A
ADDI    A,N-1
ASH     A,-K
```

ASHC Arithmetic Shift Combined



Concatenate the magnitude portions of accumulators A and $A+1$ with A on the left, and shift the 70-bit combination in bits 1–35 and 37–71 the number of places specified by E. Do not shift AC bit 0, but make bit 0 of AC $A+1$ equal to it if at least one shift occurs (*ie* if E is nonzero). If E is positive, shift left bringing 0s into bit 71 (bit 35 of AC $A+1$); bit 37 (bit 1 of AC $A+1$) is shifted into bit 35; data shifted out of bit 1 is lost; set Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative one). If E is negative, shift right bringing 0s into bit 1 if AC is positive, 1s if negative; bit 35 is shifted into bit 37; data shifted out of bit 71 is lost.

For K < 20 this is only slightly faster than IDIVI, except in the KA10 where it takes only 5–6 μs as opposed to about 16 μs for IDIVI.

Note that the effect of a shift on bit 0 of the low order word is consistent with the convention used for double length fixed point numbers. When there is no shift however, the result may be inconsistent with that convention. ▲

2.6 FLOATING POINT ARITHMETIC

For floating point arithmetic the PDP-10 has instructions for scaling the exponent (which is multiplication of the entire number by a power of 2)

In a KA10 without floating point hardware, all of the instructions presented in this section are trapped as unassigned codes [§2.10].

and negating double length numbers (software format) as well as performing addition, subtraction, multiplication and division of numbers in single precision floating point format. Moveover the KI10 has instructions for performing the four standard arithmetic operations on floating point numbers in hardware double precision format, for moving double precision numbers (with the option of taking the negative) between a pair of accumulators and a pair of memory locations, and for converting single precision numbers from fixed format to floating and vice versa. Except for the conversion instructions and the simple moves, all instructions treated here interpret all operands as floating point numbers in the formats given in §1.1, and generate results in those formats. The reader is strongly advised to reread §1.1 if he does not remember the formats in detail.

For the four standard arithmetic operations in single precision, the program can select whether or not the result shall be rounded. Rounding produces the greatest consistent precision using only single length operands. Instructions without rounding have a "long" mode, which supplies a two-word result for greater precision; the other modes save time in one-word operations where rounding is of no significance.

Actually the result is formed in a double length register in addition, subtraction and multiplication, wherein any bits of significance in the low order part supply information for normalization, and then for rounding if requested. Consider addition as an example. Before adding, the processor right shifts the fractional part of the operand with the smaller exponent until its bits correctly match the bits of the other operand in order of magnitude. Thus the smaller operand could disappear entirely, having no effect on the result ("result" shall always be taken to mean the information (one word or two) stored by the instruction, regardless of the number of significant bits it contains or even whether it is the correct answer). Long mode is likely to retain information that would otherwise be lost, but in any given mode the significance of the result depends on the relative values of the operands. Even when both operands contain twenty-seven significant bits, a long addition may store two words that together contain only one significant bit. In division the processor always calculates a one-word quotient that requires no normalization if the original operands are normalized. An extra quotient bit is calculated for rounding when requested; long mode retains the remainder.

A subtraction involving two like-signed numbers whose exponents are equal and whose fractions differ only in the LSB gives a result containing only one bit of significance.

Among the floating point instructions available only in the KI10, those that convert between number types operate only on single words. The instruction that converts to floating point assumes the operand is an integer and always normalizes and rounds the result. In the opposite direction, only the integral part of the result is saved, and rounding is an option of the program. The instructions for the four standard operations using double precision have no modes. In division the processor always calculates a two-word quotient that is normalized if the original operands are normalized, but rounding is not available. In addition, subtraction and multiplication, the result is formed in a triple length register, wherein bits of significance in the lowest order part supply information for limited normalization and then for rounding, which is automatic.

The processor has four flags, Overflow, Floating Overflow, Floating Underflow and No Divide, that indicate when the exponent is too large or

too small to be accommodated or a division cannot be performed because of the relative values of dividend and divisor. Except where the result would be in fixed point form, any of these circumstances sets Overflow and Floating Overflow. If only these two are set, the exponent of the answer is too large; if Floating Underflow is also set, the exponent is too small. No Divide being set means the processor failed to perform a division, an event that can be produced only by a zero divisor if all nonzero operands are normalized. Any condition that sets Overflow in the KI10 also sets the Trap 1 flag. These flags can be read and controlled by certain program control instructions [§§2.9, 2.10]. KI10 overflow is handled by trapping through the setting of Trap 1 [§2.9], but in the KA10, the program must make direct use of Overflow and Floating Overflow, which are available as processor conditions (via an in-out instruction) that can request a priority interrupt if enabled [§2.14]. The conditions detected can only set the arithmetic flags and the hardware does not clear them, so the program must clear them before a floating point instruction if they are to give meaningful information about the instruction afterward. However, the program can check the flags following a series of instructions to determine whether the entire series was free of the types of error detected.

In the KI10 an arithmetic instruction executed as an interrupt instruction can set no flags.

The floating point hardware functions at its best if given operands that are either normalized or zero, and except in special situations the hardware normalizes a nonzero result. An operand with a zero fraction and a nonzero exponent can give wild answers in additive operations because of extreme loss of significance; *eg* adding $\frac{1}{2} \times 2^2$ and 0×2^{69} gives a zero result, as the first operand (having a smaller exponent) looks smaller to the processor and is shifted to oblivion. A number with a 1 in bit 0 and 0s in bits 9–35 is not simply an incorrect representation of zero, but rather an unnormalized "fraction" with value −1. This unnormalized number can produce an incorrect answer in any operation. Use of other unnormalized operands simply causes loss of significant bits, except in division where they can prevent its execution because they can satisfy a no-divide condition that is impossible for normalized numbers.

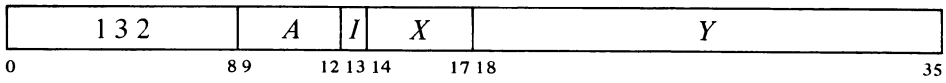
The processor normalizes the result by shifting the fraction and adjusting the exponent to compensate for the change in value. Each shift and accompanying exponent adjustment thus multiply the number both by 2 and by ½ simultaneously, leaving its value unchanged.

Note that with normalized operands, the processor uses at most two bits of information from the lowest order part to normalize the result. In multiplication this is obvious, since squaring the minimum fractional magnitude ½ gives a result of ¼. In an addition or subtraction of numbers that differ greatly in order of magnitude, the result is determined almost completely by the operand of greater order. A subtraction involving two like-signed numbers with equal exponents requires no shifting beforehand so there is no information in the lowest order part. Hence an addition or subtraction never requires shifting both before the operation and in the normalization; when there is no prior shifting, the normalization brings in 0s.

Scaling

One floating point instruction is in a category by itself: it changes the exponent of a number without changing the significance of the fraction. In other words it multiplies the number by a power of 2, and is thus analogous to arithmetic shifting of fixed point numbers except that no information is lost, although the exponent can overflow or underflow. The amount added to the exponent is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective scale factor E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the factor directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating it. A positive E increases the exponent, a negative E decreases it; E is thus the power of 2 by which the number is multiplied. The scale factor lies in the range −256 to +255.

FSC Floating Scale



This instruction can be used to float a fixed number with 27 or fewer significant bits. To float an integer contained within AC bits 9–35,

FSC AC,233

inserts the correct exponent to move the binary point from the right end to the left of bit 9 and then normalizes ($233_8 = 155_{10} = 128 + 27$).

If the fractional part of AC is zero, clear AC. Otherwise add the scale factor given by E to the exponent part of AC (thus multiplying AC by 2^E), normalize the resulting word bringing 0s into bit positions vacated at the right, and place the result back in AC.

NOTE

A negative E is represented in standard twos complement notation, but the hardware compensates for this when scaling the exponent.

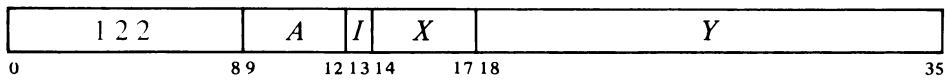
If the exponent after normalization is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

Number Conversion

In the KA10 these instructions are trapped as unassigned codes.

Although FSC can be used to float a fixed point number, the KI10 and KL10 have three single precision instructions specifically for converting between integers and floating point numbers. In all cases the operand is taken from location E, and the converted result is placed in AC.

FIX Fix



This overflow test checks for a value $\geq 2^{35}$ assuming the operand is normalized.

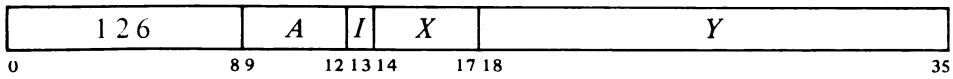
If the exponent of the floating point number in location E is > 35, set Overflow and Trap 1, and go immediately to the next instruction without affecting AC or the contents of E in any way.

Otherwise replace the exponent X in the word from location E with bits equal to the sign of the fraction, and shift the (now fixed) extended fraction $N = X - 27$ places to the correct position for its order of magnitude with the binary point at the right of bit 35. For positive N, shift left bringing 0s into bit 35 and dropping null bits out of bit 1. For negative N, shift right bringing null bits (0s for positive, 1s for negative) into bit 1, and then truncate to an integer. Place the result in AC.

This is the standard Fortran truncation ("fixation"). For it, the processor drops the

Truncation produces the integer of largest magnitude less than or equal to the magnitude of the original number. *Eg* a number $> +1$ but $< +2$ becomes $+1$; a number < -1 but > -2 becomes -1.

FIXR Fix and Round



If the exponent of the floating point number in location E is > 35, set Overflow and Trap 1, and go immediately to the next instruction without affecting AC or the contents of E in any way.

Otherwise replace the exponent X in the word from location E with bits equal to the sign of the fraction, and shift the (now fixed) extended fraction $N = X - 27$ places to the correct position for its order of magnitude with the binary point at the right of bit 35. For positive N, shift left bringing 0s into bit 35 and dropping null bits out of bit 1. For negative N, shift right bringing null bits (0s for positive, 1s for negative) into bit 1, and then round the integral part. Place the result in AC.

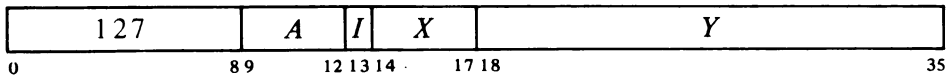
Rounding is in the positive direction: the magnitude of the integral part is increased by one if the fractional part is $\geqslant \frac{1}{2}$ in a positive number but $> \frac{1}{2}$ in a negative number. *Eg* +1.4 (decimal) is rounded to +1, whereas +1.5 and +1.6 become +2; but with negative numbers, −1.4 and −1.5 become −1, whereas −1.6 becomes −2.

fractional part in a positive number, but adds one to the integral part (as required by twos complement format) if any bits of significance are shifted out in a negative number.

This overflow test checks for a value $\geqslant 2^{35}$ assuming the operand is normalized.

This is the Algol standard for real to integer conversion. For it the processor adds one to the integral part if the fractional part is $\geqslant \frac{1}{2}$ in a positive number or (as required by twos complement format) is $\leqslant \frac{1}{2}$ in a negative number.

FLTR Float and Round



Shift the magnitude part of the fixed point integer from location E right eight places, insert the exponent 35 (in proper form) into bits 1–8 to move the shifted binary point to the left of bit 9 ($35 = 27 + 8$), and normalize the fraction bringing first the bits originally shifted out and then 0s into bit positions vacated at the right. If fewer than eight bits (left shifts) are needed to normalize, use the next bit to round the single length fraction. Place the result in AC.

The rounding function is the same as that used by the standard floating point arithmetic instructions [*see below*].

Since the largest fixed point magnitude (without considering sign) is $2^{35} - 1$, a floating point number with exponent greater than 35 (and assumed normalized) cannot be converted to fixed point. There is no limit in the opposite direction, but precision can be lost as floating point format provides fewer significant bits. A fixed integer greater than $2^{27} - 1$ cannot be represented exactly in floating point unless all its significant bits are clustered within a group of twenty-seven.

Single Precision with Rounding

In the hardware the rounding operation is actually somewhat more complex than stated here. If the result is negative, the hardware combines rounding with placing the high order word in twos complement form by decreasing its magnitude if the low order part is < ½LSB. Moreover an extra single-step renormalization occurs if the rounded word is no longer normalized.

There are four instructions that use only one-word operands and store a single-length rounded result. Rounding is away from zero: if the part of the normalized answer being dropped (the low order part of the fraction) is greater than or equal in magnitude to one half the LSB of the part being retained, the magnitude of the latter part is increased by one LSB.

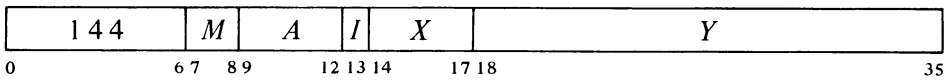
The rounding instructions have four modes that determine the source of the non-AC operand and the destination of the result. These modes are like those of logic and fixed point arithmetic, including an immediate mode that allows the instruction to carry an operand with it.

Mode	*Suffix*	*Source of non-AC operand*	*Destination of result*
Basic		*E*	AC
Immediate	I	The word *E*,0	AC
Memory	M	*E*	*E*
Both	B	*E*	AC and *E*

Note however that floating point immediate uses *E*,0 as an operand, not 0,*E*. In other words the half word *E* is interpreted as a sign, an 8-bit exponent, and a 9-bit fraction.

In each of these instructions, the exponent that results from normalization and rounding is tested for overflow or underflow. If the exponent is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < −128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

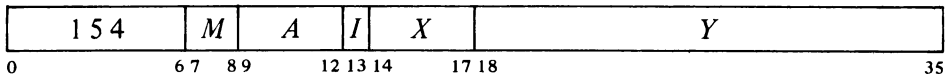
FADR **Floating Add and Round**



Floating add the operand specified by *M* to AC. If the double length fraction in the sum is zero, clear the specified destination. Otherwise normalize the double length sum bringing 0s into bit positions vacated at the right, round the high order part, test for exponent overflow or underflow as described above, and place the result in the specified destination.

FADR	Floating Add and Round	144
FADRI	Floating Add and Round Immediate	145
FADRM	Floating Add and Round to Memory	146
FADRB	Floating Add and Round to Both	147

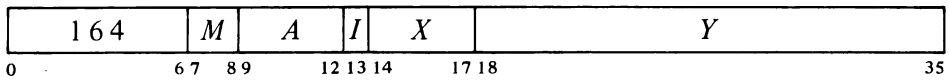
FSBR Floating Subtract and Round



Floating subtract the operand specified by *M* from AC. If the double length fraction in the difference is zero, clear the specified destination. Otherwise normalize the double length difference bringing 0s into bit positions vacated at the right, round the high order part, test for exponent overflow or underflow as described above, and place the result in the specified destination.

FSBR	Floating Subtract and Round	154
FSBRI	Floating Subtract and Round Immediate	155
FSBRM	Floating Subtract and Round to Memory	156
FSBRB	Floating Subtract and Round to Both	157

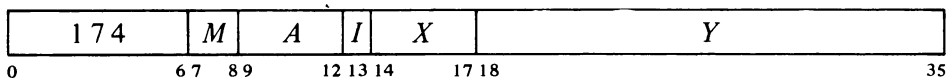
FMPR Floating Multiply and Round



Floating Multiply AC by the operand specified by *M*. If the double length fraction in the product is zero, clear the specified destination. Otherwise normalize the double length product bringing 0s into bit positions vacated at the right, round the high order part, test for exponent overflow or underflow as described above, and place the result in the specified destination.

FMPR	Floating Multiply and Round	164
FMPRI	Floating Multiply and Round Immediate	165
FMPRM	Floating Multiply and Round to Memory	166
FMPRB	Floating Multiply and Round to Both	167

FDVR Floating Divide and Round



If the magnitude of the fraction in AC is greater than or equal to twice that of the fraction in the operand specified by *M*, set Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If the division can be performed, floating divide AC by the operand specified by *M*, calculating a quotient fraction of 28 bits (this includes an extra bit for rounding). If the fraction is zero, clear the specified destination. Otherwise round the fraction using the extra bit calculated. If the original

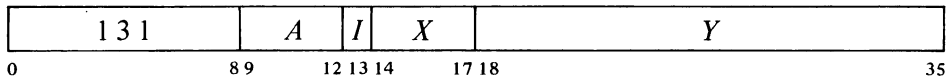
operands were normalized, the single length quotient will already be normalized; if it is not, normalize it bringing 0s into bit positions vacated at the right. Test for exponent overflow or underflow as described above, and place the result in the specified destination.

FDVR	Floating Divide and Round	174
FDVRI	Floating Divide and Round Immediate	175
FDVRM	Floating Divide and Round to Memory	176
FDVRB	Floating Divide and Round to Both	177

Single Precision without Rounding

Instructions that do not round are faster for processing floating point numbers with fractions containing fewer than 27 significant bits. On the other hand the long mode provides double precision (software format) or allows the programmer to use his own method of rounding. Besides the four usual arithmetic operations with normalization, there are two nonnormalizing instructions that facilitate software double precision arithmetic [§2.11 *gives examples of double precision floating point routines*]. These two instructions have no modes.

DFN **Double Floating Negate**



Negate the double length floating point number composed of the contents of AC and location *E* with AC on the left. Do this by taking the twos complement of the number whose sign is AC bit 0, whose exponent is in AC bits 1–8, and whose fraction is the 54-bit string in bits 9–35 of AC and location *E*. Place the high order word of the result in AC; place the low order part of the fraction in bits 9–35 of location *E* without altering the original contents of bits 0–8 of that location.

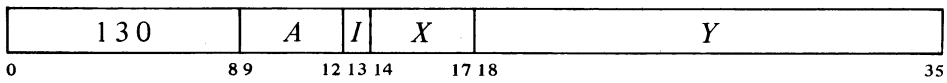
Note that this instruction can be used to negate numbers in software double precision format only; for the KI10 hardware double precision format, the program must use the double moves.

Usually the double length number is in two adjacent accumulators, and *E* equals *A*+1. There is no overflow test, as negating a correctly formatted floating point number cannot cause overflow.

DFN AC,AC is undefined.

At the Stanford A.I. Lab and at LOTS, the DFN instruction is simulated in software.

UFA **Unnormalized Floating Add**



Floating add the contents of location *E* to AC. If the double length fraction in the sum is zero, clear accumulator *A*+1. Otherwise normalize the sum only if the magnitude of its fractional part is ≥ 1, and place the high order part of the result in AC *A*+1. The original contents of AC and *E* are unaffected.

The caution given below for FAD applies to this instruction as well.

NOTE

The result is placed in accumulator $A+1$. This is the only arithmetic instruction that stores the result in a second accumulator, leaving the original operands intact.

The exponent of the sum is equal to that of the larger summand unless addition of the fractions overflows, in which case it is greater by 1. Exponent overflow can occur only in the latter case.

If the exponent of the sum following the one-step normalization is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one.

At the Stanford A.I. Lab KL10 and at LOTS the UFA instruction is simulated in software.

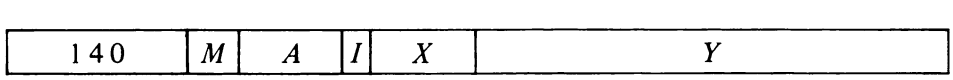
The remaining single precision floating point instructions perform the four standard arithmetic operations with normalization but without rounding. All use AC and the contents of location E as operands and have four modes.

Mode	*Suffix*	*Effect*
Basic		High order word of result stored in AC.
Long	L	In addition, subtraction and multiplication, the two-word result (in the software double length format described in §1.1) is stored in accumulators A and $A+1$. In division the dividend is the double length word in A and $A+1$; the single length quotient is stored in AC, the remainder in AC $A+1$.
Memory	M	High order word of result stored in E.
Both	B	High order word of result stored in AC and E.

The Stanford A.I.Lab KL10 and the LOTS 2040 simulate all Long mode instructions in software.

In each of these instructions, the exponent that results from normalization is tested for overflow or underflow. If the exponent is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

FAD **Floating Add**



Floating add the contents of location E to AC. If the double length fraction in the sum is zero, clear the destination specified by M, clearing both accumulators in long mode. Otherwise normalize the double length sum bringing 0s into bit positions vacated at the right, test for exponent overflow or

CAUTION

In single precision addition the term with the smaller exponent is right shifted in a double

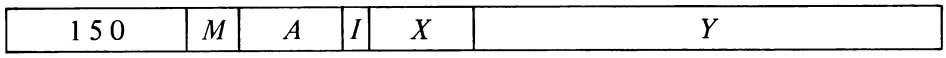
length register, specifically a register with 54 magnitude bits. Now if the difference in the exponents is < 54, there is at least one significant bit after the shift (assuming normalized operands); and if the difference is > 64, the hardware throws the term away by substituting zero. But when the exponent difference lies in the range 54 to 64, the procedure disposes of all significant bits without actually substituting zero. This means that if the shifted term is positive it appears in the addition as all 0s, but if negative it appears as all 1s. The latter case gives an answer that is less by one LSB.

underflow as described above, and place the high order word of the result in the specified destination.

▲ In long mode if the exponent of the sum is < -101 $(-128 + 27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the sum in bits 1–8, and the low order part of the fraction in bits 9–35.

FAD	Floating Add	140
FADL	Floating Add Long	141
FADM	Floating Add to Memory	142
FADB	Floating Add to Both	143

FSB **Floating Subtract**



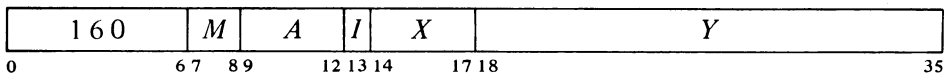
The caution given above for addition applies also to subtraction, which is done by adding with the minuend negated. Here the lesser answer (as against a true zero substitution) occurs when the term with the smaller exponent is negative after the minuend negation, *ie* when it is a negative subtrahend but a positive minuend.

Floating subtract the contents of location E from AC. If the double length fraction in the difference is zero, clear the destination specified by M, clearing both accumulators in long mode. Otherwise normalize the double length difference bringing 0s into bit positions vacated at the right, test for exponent overflow or underflow as described above, and place the high order word of the result in the specified destination.

▲ In long mode if the exponent of the difference is < -101 $(-128 + 27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the difference in bits 1–8, and the low order part of the fraction in bits 9–35.

FSB	Floating Subtract	150
FSBL	Floating Subtract Long	151
FSBM	Floating Subtract to Memory	152
FSBB	Floating Subtract to Both	153

FMP **Floating Multiply**



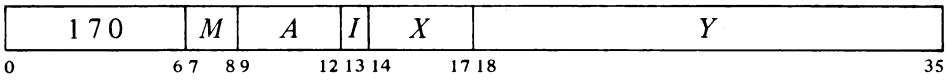
Floating multiply AC by the contents of location E. If the double length fraction in the product is zero, clear the destination specified by M, clearing both accumulators in long mode. Otherwise normalize the double length

product bringing 0s into bit positions vacated at the right, test for exponent overflow or underflow as described above, and place the high order word of the result in the specified destination.

In long mode if the exponent of the product is > 154 $(127+27)$ or < -101 $(-128+27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the product in bits 1–8, and the low order part of the fraction in bits 9–35.

FMP	Floating Multiply	160
FMPL	Floating Multiply Long	161
FMPM	Floating Multiply to Memory	162
FMPB	Floating Multiply to Both	163

FDV Floating Divide



If the magnitude of the fraction in AC (in long mode, the high order part of the magnitude of the double length fraction in accumulators A and $A+1$) is greater than or equal to twice the magnitude of the fraction in location E, set Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way. ▲

Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If division can be performed, floating divide the AC operand by the contents of location E. In long mode the AC operand (the dividend) is the double length number in accumulators A and $A+1$; in other modes it is the single word in AC. Calculate a quotient fraction of 27 bits. If the fraction is zero, clear the destination specified by M, clearing both accumulators in long mode if the double length dividend was zero. A quotient with a nonzero fraction will already be normalized if the original operands were normalized; if it is not, normalize it bringing 0s into bit positions vacated at the right. Test for exponent overflow or underflow as described above, and place the single length quotient part of the result in the specified destination.

In long mode a nonzero unnormalized dividend whose entire high order fraction is zero produces a zero quotient. In this case the second AC is cleared in the KI10 but may receive rubbish in the KA10.

In long mode calculate the exponent for the fractional remainder from the division according to the relative magnitudes of the fractions in dividend and divisor: if the dividend was greater than or equal to the divisor, the exponent of the remainder is 26 less than that of the dividend, otherwise it is 27 less. If the remainder exponent is < -128 or the fraction is zero, clear AC $A+1$. Otherwise place the floating point remainder (exponent and fraction) with the sign of the dividend in AC $A+1$.

FDV	Floating Divide	170
FDVL	Floating Divide Long	171
FDVM	Floating Divide to Memory	172
FDVB	Floating Divide to Both	173

Double Precision Operations

In the KA10 these instructions are trapped as unassigned codes.

Although double precision floating point arithmetic can be done by routines using the single precision instructions and the software double length format, the KI10 has instructions specifically for handling double length operands in the hardware double precision format described in §1.1. Four of the instructions use two double length operands, perform the standard arithmetic operations, and store double length results. The other four instructions each move one double length operand between the accumulators and memory, either unchanged or negated.

All of these instructions address a pair of adjacent accumulators and a pair of adjacent memory locations. The accumulators have addresses A and $A+1$ (mod 20_8) just as they do for the double length operands used in some shift, rotate and single precision arithmetic instructions. The memory locations have addresses E and $E+1$ (mod 2^{18}), where the second address is 0 if E is 777777.

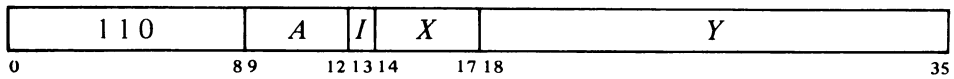
For the two instructions that simply move a pair of words without altering them, the format of those words is actually irrelevant. The other six instructions process each word pair as a double length number in the hardware floating point format. Hence they ignore bit 0 in the low order word of every operand and clear that bit in the result.

The four nonmove instructions perform the standard arithmetic operations. All use two double length operands in the hardware double precision format, one from the accumulators and one from memory. In the KI10, addition and subtraction always normalize the result; in multiplication and division the result is guaranteed to be normalized only if the original operands are normalized. In the KL10, all operations return normalized results whether the original operands were normalized or not. In all cases the result, rounded except in KI10 division, is placed in the accumulators. The rounding function is the same as that used in single precision: if the part of the answer being dropped (the low order part of the fraction) is greater than or equal in magnitude to one half the LSB of the double length part being retained, the magnitude of the latter part is increased by one LSB (with appropriate adjustment for a twos complement negative).

An arithmetic instruction executed as an interrupt instruction can set no flags.

In each of these instructions, the exponent that results from normalization and rounding (if done) is tested for overflow or underflow. If the exponent is > 127, set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128, set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one. Setting Overflow also sets the Trap 1 flag.

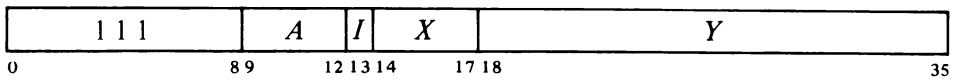
DFAD Double Floating Add



Floating add the operand of locations E and $E+1$ to the operand of accumulators A and $A+1$. If the fraction in the sum is zero, clear A and $A+1$.

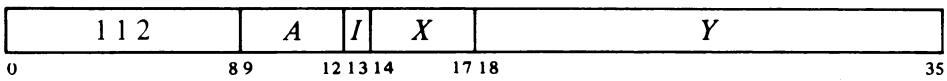
Otherwise normalize the triple length sum bringing 0s in at the right, round the high order double length part, test for exponent overflow or underflow as described above, and place the result in ACs A and A+1.

DFSB Double Floating Subtract



Floating subtract the operand of locations E and E+1 from the operand of accumulators A and A+1. If the fraction in the difference is zero, clear A and A+1. Otherwise normalize the triple length difference bringing 0s into bit positions vacated at the right, round the high order double length part, test for exponent overflow or underflow as described above, and place the result in ACs A and A+1.

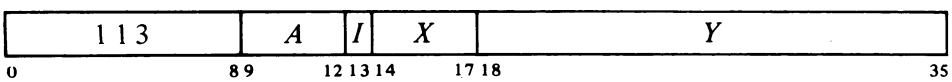
DFMP Double Floating Multiply



Floating multiply the operand of accumulators A and A+1 by the operand of locations E and E+1. If the product is zero, clear A and A+1. Otherwise, normalize and round the product, test for exponent overflow and underflow as described above, and place the result in ACs A and A+1.

The KI10 normalizes, at most, one place. This produces a normalized result for normalized operands.

DFDV Double Floating Divide



If the magnitude of the fraction in the operand of accumulators A and A+1 is greater than or equal to twice that of the fraction in the operand of locations E and E+1, set Overflow, Floating Overflow, No Divide and Trap 1, and go immediately to the next instruction without affecting the original AC or memory operands in any way.

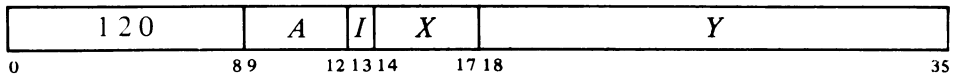
Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If the division can be performed, floating divide the AC operand by the memory operand, calculating a quotient fraction. If the fraction is zero, clear

A nonzero quotient is normalized if the original operands are normalized.

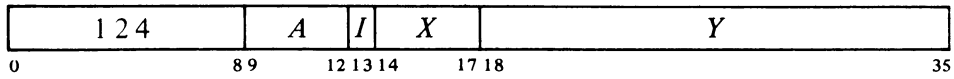
A and $A+1$. Otherwise round (KL10 only), and test for exponent overflow or underflow as described above, and place the double length quotient part of the result in ACs A and $A+1$ (the remainder is lost).

DMOVE Double Move



Move the contents of locations E and $E+1$ respectively to accumulators A and $A+1$. The memory locations are unaffected, the original contents of the ACs are lost.

DMOVEM Double Move to Memory



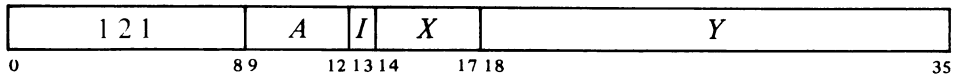
Move the contents of accumulators A and $A+1$ respectively to locations E and $E+1$. The ACs are unaffected, the original contents of the memory locations are lost.

Do not use the instruction DMOVEM AC,AC+1. The KI10 places AC in both AC+1 and AC+2.

In the KL10, a copy of AC+1 is placed in AC+2 and a copy of AC is placed in AC+1.

Use DMOVE and DMOVN only for double transfers between ACs.

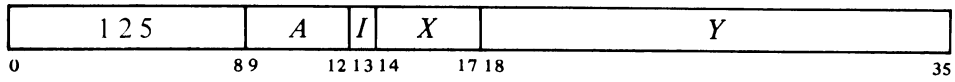
DMOVN Double Move Negative



Negate the double length floating point number taken from locations E and $E+1$, and move it to accumulators A and $A+1$. The memory locations are unaffected, the original contents of the ACs are lost.

Note also that on the KI10 there is no overflow test, as negating a correctly formatted floating point number cannot cause overflow. However, on the KL10, DMOVN, DMOVNM may be used for handling double precision integers and if -2^{70} is negated, overflow will occur, setting Overflow, Carry 1, and Trap 1 flags. If the source operand is zero, Carry 0 and Carry 1 will be set.

DMOVNM Double Move Negative to Memory



Negate the double length floating point number taken from accumulators A and $A+1$, and move it to locations E and $E+1$. The ACs are unaffected, the original contents of the memory locations are lost.

Do not use the instruction DMOVNM AC, AC+1. At

Although the configuration of the operands is irrelevant in DMOVE and DMOVEM, none of the above instructions is available in the KA10. Therefore unless a program is actually doing floating point arithmetic in the hardware double precision format, it is recommended that the double moves not be used in KI10 programs so they will be compatible with the KA10. Simply to move a two-word operand unaltered requires two one-word moves. To negate a two-word operand that is actually in the hardware format requires a somewhat longer substitution; *eg* this sequence is equivalent to DMOVN AC,E.

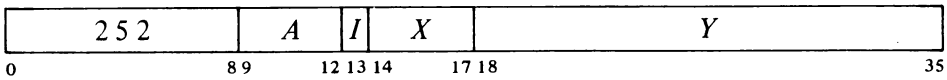
present the KI10 places the negative of AC (the complement, if AC+1 originally contains zero) into AC+1, and the negative of that into AC+2, but this result is not guaranteed.

```
SETCM   AC,E            ;Take ones complement of high word
MOVN    AC+1,E+1        ;Take twos complement of low word
TLZ     AC+1,400000         ;Clear bit 0
SKIPN   AC+1            ;If low part is zero, change high word
ADDI    AC,1            ;to twos complement
```

2.7 ARITHMETIC TESTING

These instructions may jump or skip depending on the result of an arithmetic test and may first perform an arithmetic operation on the test word. Two of the instructions have no modes.

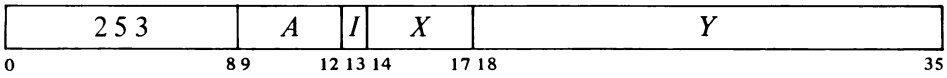
AOBJP Add One to Both Halves of AC and Jump if Positive



Add one to each half of AC and place the result back in AC. If the result is greater than or equal to zero (*ie* if bit 0 is 0, and hence a negative count in the left half has reached zero or a positive count has not yet reached 2^{17}), take the next instruction from location *E* and continue sequential operation from there.

Note: The KA10 increments the two halves of AC by adding $1\,000001_8$ to the entire register. In the KI10 the two halves are handled independently.

AOBJN Add One to Both Halves of AC and Jump if Negative



Add one to each half of AC and place the result back in AC. If the result is less than zero (*ie* if bit 0 is 1, and hence a negative count in the left half has not yet reached zero or a positive count has reached 2^{17}), take the next instruction from location *E* and continue sequential operation from there.

Note

The KA10 increments the two halves of AC by adding $1\,000001_8$ to the entire register. In the KI10 and KL10 the two halves are handled independently.

In the KA10, incrementing both halves of AC together is effected by adding $1\,000001_8$. A count of -2 in AC left is therefore increased to zero if $2^{18} - 1$ is incremented in AC right.

These two instructions allow the program to keep a control count in the left half of an index register and require only one data transfer to initialize. Problem: Add 3 to each location in a table of N entries starting at TAB. Only four instructions are required.

```
MOVSI   XR,-N          ;Put -N in XR left (clear XR right)
MOVEI   AC,3           ;Put 3 in AC
ADDM    AC,TAB(XR)     ;Add 3 to entry
AOBJN   XR,.-1         ;Update XR and go back unless all
                       ;entries accounted for
```

The eight remaining instructions jump or skip if the operand or operands satisfy a test condition specified by the mode.

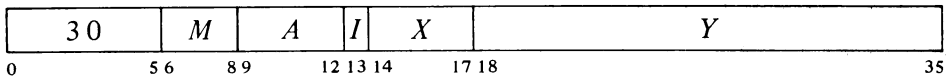
Mode	*Suffix*
Never	
Less	L
Equal	E
Less or Equal	LE
Always	A
Greater or Equal	GE
Not Equal	N
Greater	G

Instructions with one operand compare AC or the contents of location E with zero, those with two compare AC with E or the contents of location E. The processor always makes the comparison even though the result is used in only six of the modes. If the mnemonic has no suffix there is never any program control function, and the instruction may be a no-op; an A suffix produces an unconditional jump or skip – the action is always taken regardless of how the two quantities compare.

In the KI10 an arithmetic instruction executed as an interrupt instruction can set no flags.

The last four of these instructions perform arithmetic operations, which are checked for overflow. In the KI10 and KL10 any condition that sets Overflow also sets the Trap 1 flag.

CAI Compare AC Immediate and Skip if Condition Satisfied

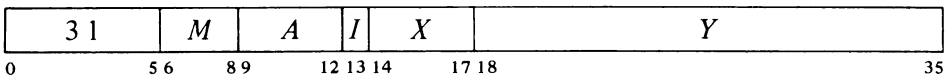


Compare AC with E (*ie* with the word $0,E$) and skip the next instruction in sequence if the condition specified by M is satisfied.

CAI	Compare AC Immediate but Do Not Skip	300
CAIL	Compare AC Immediate and Skip if AC Less than E	301
CAIE	Compare AC Immediate and Skip if Equal	302
CAILE	Compare AC Immediate and Skip if AC Less than or Equal to E	303
CAIA	Compare AC Immediate but Always Skip	304
CAIGE	Compare AC Immediate and Skip if AC Greater than or Equal to E	305
CAIN	Compare AC Immediate and Skip if Not Equal	306
CAIG	Compare AC Immediate and Skip if AC Greater than E	307

CAI is a no-op in which I, X and Y are reserved for future use and should be zero (at present E is ignored).

CAM Compare AC with Memory and Skip if Condition Satisfied

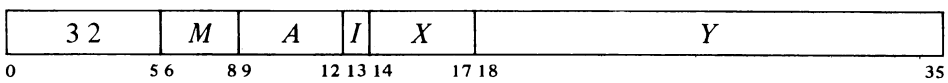


Compare AC with the contents of location E and skip the next instruction in sequence if the condition specified by M is satisfied. The pair of numbers compared may be either both fixed or both normalized floating point.

CAM	Compare AC with Memory but Do Not Skip	310
CAML	Compare AC with Memory and Skip if AC Less	311
CAME	Compare AC with Memory and Skip if Equal	312
CAMLE	Compare AC with Memory and Skip if AC Less or Equal	313
CAMA	Compare AC with Memory but Always Skip	314
CAMGE	Compare AC with Memory and Skip if AC Greater or Equal	315
CAMN	Compare AC with Memory and Skip if Not Equal	316
CAMG	Compare AC with Memory and Skip if AC Greater	317

CAM is a no-op that references memory.

JUMP Jump if AC Condition Satisfied



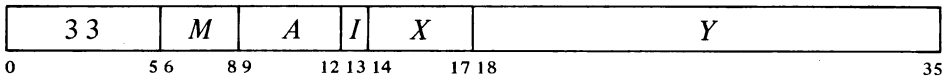
Compare AC (fixed or floating) with zero, and if the condition specified by

M is satisfied, take the next instruction from location *E* and continue sequential operation from there.

JUMP is a no-op (instruction code 320 has this mnemonic for symmetry). In it, *I*, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored).

JUMP	Do Not Jump	320
JUMPL	Jump if AC Less than Zero	321
JUMPE	Jump if AC Equal to Zero	322
JUMPLE	Jump if AC Less than or Equal to Zero	323
JUMPA	Jump Always	324
JUMPGE	Jump if AC Greater than or Equal to Zero	325
JUMPN	Jump if AC Not Equal to Zero	326
JUMPG	Jump if AC Greater than Zero	327

SKIP Skip if Memory Condition Satisfied



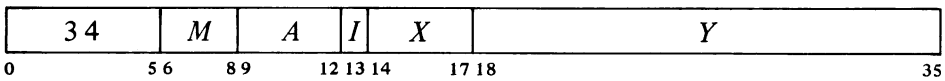
If *A* is zero, SKIP is a no-op; otherwise it is equivalent to MOVE. (Instruction code 330 has mnemonic SKIP for symmetry.)

Compare the contents (fixed or floating) of location *E* with zero, and skip the next instruction in sequence if the condition specified by *M* is satisfied. If *A* is nonzero also place the contents of location *E* in AC.

SKIP	Do Not Skip	330
SKIPL	Skip if Memory Less than Zero	331
SKIPE	Skip if Memory Equal to Zero	332
SKIPLE	Skip if Memory Less than or Equal to Zero	333
SKIPA	Skip Always	334
SKIPGE	Skip if Memory Greater than or Equal to Zero	335
SKIPN	Skip if Memory Not Equal to Zero	336
SKIPG	Skip if Memory Greater than Zero	337

SKIPA is a convenient way to load an accumulator and skip over an instruction upon entering a loop.

AOJ Add One to AC and Jump if Condition Satisfied

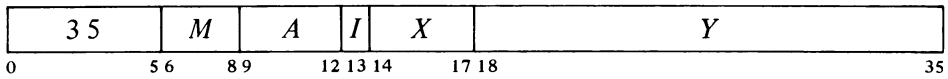


Increment AC by one and place the result back in AC. Compare the result with zero, and if the condition specified by *M* is satisfied, take the next instruction from location *E* and continue sequential operation from there. If AC originally contained $2^{35} - 1$, set the Overflow and Carry 1 flags; if -1, set Carry 0 and Carry 1.

AOJ	Add One to AC but Do Not Jump	340
AOJL	Add One to AC and Jump if Less than Zero	341
AOJE	Add One to AC and Jump if Equal to Zero	342
AOJLE	Add One to AC and Jump if Less than or Equal to Zero	343

AOJA	Add One to AC and Jump Always	344
AOJGE	Add One to AC and Jump if Greater than or Equal to Zero	345
AOJN	Add One to AC and Jump if Not Equal to Zero	346
AOJG	Add One to AC and Jump if Greater than Zero	347

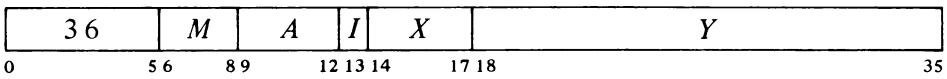
AOS Add One to Memory and Skip if Condition Satisfied



Increment the contents of location E by one and place the result back in E. Compare the result with zero, and skip the next instruction in sequence if the condition specified by M is satisfied. If location E originally contained $2^{35} - 1$, set the Overflow and Carry 1 flags; if -1, set Carry 0 and Carry 1. If A is nonzero also place the result in AC.

AOS	Add One to Memory but Do Not Skip	350
AOSL	Add One to Memory and Skip if Less than Zero	351
AOSE	Add One to Memory and Skip if Equal to Zero	352
AOSLE	Add One to Memory and Skip if Less than or Equal to Zero	353
AOSA	Add One to Memory and Skip Always	354
AOSGE	Add One to Memory and Skip if Greater than or Equal to Zero	355
AOSN	Add One to Memory and Skip if Not Equal to Zero	356
AOSG	Add One to Memory and Skip if Greater than Zero	357

SOJ Subtract One from AC and Jump if Condition Satisfied

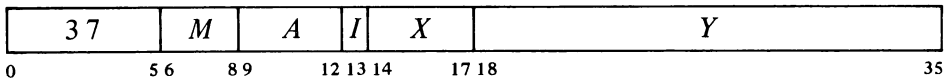


Decrement AC by one and place the result back in AC. Compare the result with zero, and if the condition specified by M is satisfied, take the next instruction from location E and continue sequential operation from there. If AC originally contained -2^{35}, set the Overflow and Carry 0 flags; if any other nonzero number, set Carry 0 and Carry 1.

SOJ	Subtract One from AC but Do Not Jump	360
SOJL	Subtract One from AC and Jump if Less than Zero	361
SOJE	Subtract One from AC and Jump if Equal to Zero	362
SOJLE	Subtract One from AC and Jump if Less than or Equal to Zero	363

SOJA	Subtract One from AC and Jump Always	364
SOJGE	Subtract One from AC and Jump if Greater than or Equal to Zero	365
SOJN	Subtract One from AC and Jump if Not Equal to Zero	366
SOJG	Subtract One from AC and Jump if Greater than Zero	367

SOS Subtract One from Memory and Skip if Condition Satisfied



Decrement the contents of location E by one and place the result back in E. Compare the result with zero, and skip the next instruction in sequence if the condition specified by M is satisfied. If location E originally contained -2^{35}, set the Overflow and Carry 0 flags; if any other nonzero number, set Carry 0 and Carry 1. If A is nonzero also place the result in AC.

SOS	Subtract One from Memory but Do Not Skip	370
SOSL	Subtract One from Memory and Skip if Less than Zero	371
SOSE	Subtract One from Memory and Skip if Equal to Zero	372
SOSLE	Subtract One from Memory and Skip if Less than or Equal to Zero	373
SOSA	Subtract One from Memory and Skip Always	374
SOSGE	Subtract One from Memory and Skip if Greater than or Equal to Zero	375
SOSN	Subtract One from Memory and Skip if Not Equal to Zero	376
SOSG	Subtract One from Memory and Skip if Greater than Zero	377

Some of these instructions are useful for determining the relative values of fixed and floating point numbers; others are convenient for controlling iterative processes by counting. AOSE is especially useful in an interlock procedure in a multiprocessor system. Suppose memory contains a routine that must be available to two processors but cannot be used by both at once. When one processor finishes the routine it sets location LOCK to -1. Either processor can then test the interlock and make it busy with no possibility of letting the other one in, as AOSE cannot be interrupted once it starts to modify the addressed location.

This procedure is invalid in the KA10 if the programmer

```
AOSE    LOCK    ;Skip to interlocked code only if
JRST    .-1     ;LOCK is zero after addition
.               ;Interlocked code starts here
.
.
SETOM   LOCK    ;Unlock
```

is making use of the drum split feature (which is not used by any DEC equipment).

Since it takes several days to count to 2^{36}, it is alright to keep testing the lock.

2.8 LOGICAL TESTING AND MODIFICATION

These eight instructions use a mask to modify and/or test selected bits in AC. The bits are those that correspond to 1s in the mask and they are referred to as the "masked bits". The programmer chooses the mask, the way in which the masked bits are to be modified, and the condition the masked bits must satisfy to produce a skip.

The basic mnemonics are three letters beginning with T. The second letter selects the mask and the manner in which it is used.

Mask	*Letter*	*Effect*
Right	R	AC right is masked by *E* (AC is masked by the word 0,*E*)
Left	L	AC left is masked by *E* (AC is masked by the word *E*,0)
Direct	D	AC is masked by the contents of location *E*
Swapped	S	AC is masked by the contents of location *E* with left and right halves interchanged

The third letter determines the way in which those bits selected by the mask are modified.

Modification	*Letter*	*Effect on AC*
No	N	None
Zeros	Z	Places 0s in all masked bit positions
Complement	C	Complements all masked bits
Ones	O	Places 1s in all masked bit positions

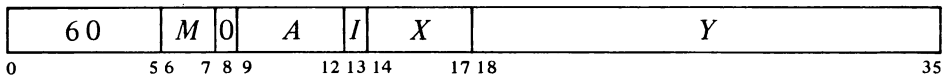
An additional letter may be appended to indicate the mode, which specifies the condition the masked bits must satisfy to produce a skip.

These mode names are consistent with those for arithmetic testing and derive from the test method, which ands AC with the mask and tests whether the result is equal to zero or is not equal to zero. The programmer may find it convenient to think of the modes as Every and Not Every: every masked bit is 0 or not every masked bit is 0.

Mode	*Suffix*	*Effect*
Never		Never skip
Equal	E	Skip if all masked bits equal 0
Always	A	Always skip
Not Equal	N	Skip if not all masked bits equal 0 (at least one bit is 1)

If the mnemonic has no suffix there is never any skip, and the instruction is a no-op if there is also no modification; an A suffix produces an unconditional skip – the skip always occurs regardless of the state of the masked bits. Note that the skip condition must be satisfied by the state of the masked bits *prior* to any modification called for by the instruction.

TRN Test Right, No Modification, and Skip if Condition Satisfied

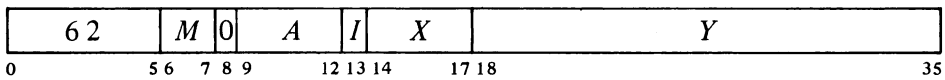


If the bits in AC right corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

TRN is a no-op in which *I*, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored).

TRN	Test Right, No Modification, but Do Not Skip	600
TRNE	Test Right, No Modification, and Skip if All Masked Bits Equal 0	602
TRNA	Test Right, No Modification, but Always Skip	604
TRNN	Test Right, No Modification, and Skip if Not All Masked Bits Equal 0	606

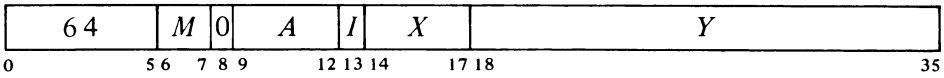
TRZ Test Right, Zeros, and Skip if Condition Satisfied



If the bits in AC right corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TRZ	Test Right, Zeros, but Do Not Skip	620
TRZE	Test Right, Zeros, and Skip if All Masked Bits Equaled 0	622
TRZA	Test Right, Zeros, but Always Skip	624
TRZN	Test Right, Zeros, and Skip if Not All Masked Bits Equaled 0	626

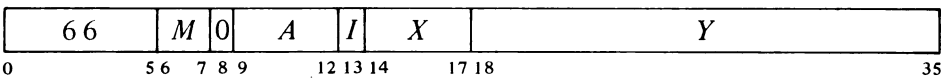
TRC Test Right, Complement, and Skip if Condition Satisfied



If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

TRC	Test Right, Complement, but Do Not Skip	640
TRCE	Test Right, Complement, and Skip if All Masked Bits Equaled 0	642
TRCA	Test Right, Complement, but Always Skip	644
TRCN	Test Right, Complement, and Skip if Not All Masked Bits Equaled 0	646

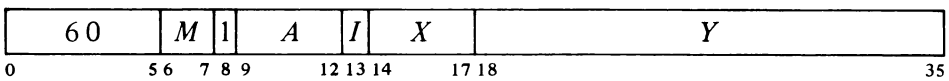
TRO Test Right, Ones, and Skip if Condition Satisfied



If the bits in AC right corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TRO	Test Right, Ones, but Do Not Skip	660
TROE	Test Right, Ones, and Skip if All Masked Bits Equaled 0	662
TROA	Test Right, Ones, but Always Skip	664
TRON	Test Right, Ones, and Skip if Not All Masked Bits Equaled 0	666

TLN Test Left, No Modification, and Skip if Condition Satisfied

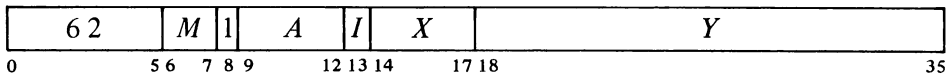


If the bits in AC left corresponding to 1s in E satisfy the condition specified by M, skip the next instruction in sequence. AC is unaffected.

TLN	Test Left, No Modification, but Do Not Skip	601
TLNE	Test Left, No Modification, and Skip if All Masked Bits Equal 0	603
TLNA	Test Left, No Modification, but Always Skip	605
TLNN	Test Left, No Modification, and Skip if Not All Masked Bits Equal 0	607

TLN is a no-op in which I, X and Y are reserved for future use and should be zero (at present E is ignored).

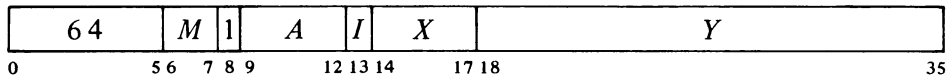
TLZ Test Left, Zeros, and Skip if Condition Satisfied



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TLZ	Test Left, Zeros, but Do Not Skip	621
TLZE	Test Left, Zeros, and Skip if All Masked Bits Equaled 0	623
TLZA	Test Left, Zeros, but Always Skip	625
TLZN	Test Left, Zeros, and Skip if Not All Masked Bits Equaled 0	627

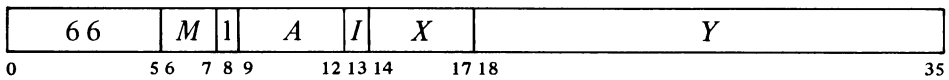
TLC Test Left, Complement, and Skip if Condition Satisfied



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

TLC	Test Left, Complement, but Do Not Skip	641
TLCE	Test Left, Complement, and Skip if All Masked Bits Equaled 0	643
TLCA	Test Left, Complement, but Always Skip	645
TLCN	Test Left, Complement, and Skip if Not All Masked Bits Equaled 0	647

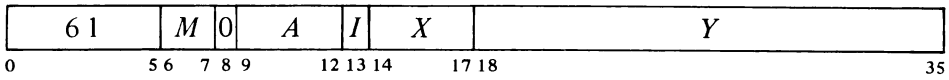
TLO Test Left, Ones, and Skip if Condition Satisfied



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TLO	Test Left, Ones, but Do Not Skip	661
TLOE	Test Left, Ones, and Skip if All Masked Bits Equaled 0	663
TLOA	Test Left, Ones, but Always Skip	665
TLON	Test Left, Ones, and Skip if Not All Masked Bits Equaled 0	667

TDN Test Direct, No Modification, and Skip if Condition Satisfied

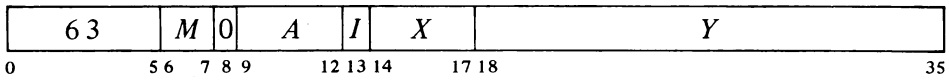


If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

TDN	Test Direct, No Modification, but Do Not Skip	610
TDNE	Test Direct, No Modification, and Skip if All Masked Bits Equal 0	612
TDNA	Test Direct, No Modification, but Always Skip	614
TDNN	Test Direct, No Modification, and Skip if Not All Masked Bits Equal 0	616

TDN is a no-op that references memory.

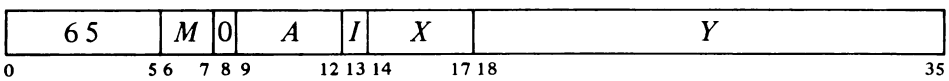
TDZ Test Direct, Zeros, and Skip if Condition Satisfied



If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TDZ	Test Direct, Zeros, but Do Not Skip	630
TDZE	Test Direct, Zeros, and Skip if All Masked Bits Equaled 0	632
TDZA	Test Direct, Zeros, but Always Skip	634
TDZN	Test Direct, Zeros, and Skip if Not All Masked Bits Equaled 0	636

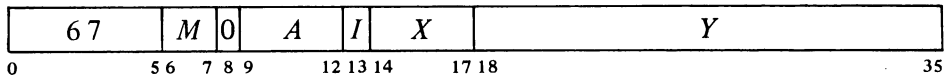
TDC Test Direct, Complement, and Skip if Condition Satisfied



If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

TDC	Test Direct, Complement, but Do Not Skip	650
TDCE	Test Direct, Complement, and Skip if All Masked Bits Equaled 0	652
TDCA	Test Direct, Complement, but Always Skip	654
TDCN	Test Direct, Complement, and Skip if Not All Masked Bits Equaled 0	656

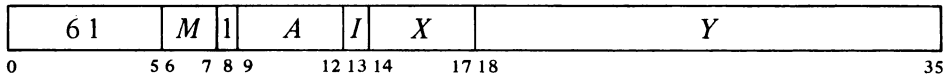
TDO Test Direct, Ones, and Skip if Condition Satisfied



If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TDO	Test Direct, Ones, but Do Not Skip	670
TDOE	Test Direct, Ones, and Skip if All Masked Bits Equaled 0	672
TDOA	Test Direct, Ones, but Always Skip	674
TDON	Test Direct, Ones, and Skip if Not All Masked Bits Equaled 0	676

TSN Test Swapped, No Modification, and Skip if Condition Satisfied

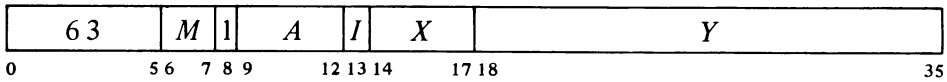


If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

TSN is a no-op that references memory.

TSN	Test Swapped, No Modification, but Do Not Skip	611
TSNE	Test Swapped, No Modification, and Skip if All Masked Bits Equal 0	613
TSNA	Test Swapped, No Modification, but Always Skip	615
TSNN	Test Swapped, No Modification, and Skip if Not All Masked Bits Equal 0	617

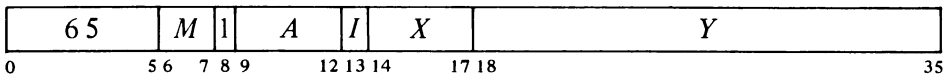
TSZ Test Swapped, Zeros, and Skip if Condition Satisfied



If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TSZ	Test Swapped, Zeros, but Do Not Skip	631
TSZE	Test Swapped, Zeros, and Skip if All Masked Bits Equaled 0	633
TSZA	Test Swapped, Zeros, but Always Skip	635
TSZN	Test Swapped, Zeros, and Skip if Not All Masked Bits Equaled 0	637

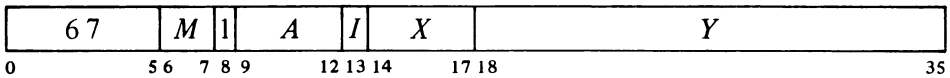
TSC **Test Swapped, Complement, and Skip if Condition Satisfied**



If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

TSC	Test Swapped, Complement, but Do Not Skip	651
TSCE	Test Swapped, Complement, and Skip if All Masked Bits Equaled 0	653
TSCA	Test Swapped, Complement, but Always Skip	655
TSCN	Test Swapped, Complement, and Skip if Not All Masked Bits Equaled 0	657

TSO **Test Swapped, Ones, and Skip if Condition Satisfied**



If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TSO	Test Swapped, Ones, but Do Not Skip	671
TSOE	Test Swapped, Ones, and Skip if All Masked Bits Equaled 0	673
TSOA	Test Swapped, Ones, but Always Skip	675
TSON	Test Swapped, Ones, and Skip if Not All Masked Bits Equaled 0	677

With these instructions any bit throughout all of memory can be used as a program flag, although an ordinary memory location containing flags must be moved to an accumulator for testing or modification. The usual procedure, since locations 1–17 are addressable as index registers, is to use AC 0 as a register of flags (often addressed symbolically as F).

Unless one frequently tests flags in both halves of F simultaneously, it is generally most convenient to select bits by 1s right in the address part of the instruction word. A given bit selected by a half word mask *M* is then set by one of these:

TRO F,*M* TLO F,*M*

and tested and cleared by one of these:

TRZE F,*M* TRZN F,*M* TLZE F,*M* TLZN F,*M*

Suppose we wish to skip if both bits 34 and 35 are 1 in location L. The following suffices.

```
SETCM   F,L
TRNE    F,3
```

We can refer to a flag in a given bit position within a word as flag *X*, where *X* is a binary number containing a single 1 in the same bit position as the flag. This sequence determines whether flags *X* and *Y* in the right half of accumulator F are both on:

```
TRC     F,X+Y     ;Complement flags X and Y
TRCE    F,X+Y     ;Test both and restore original states
...               ;Do this if not both on
...               ;Skip to here if both on
```

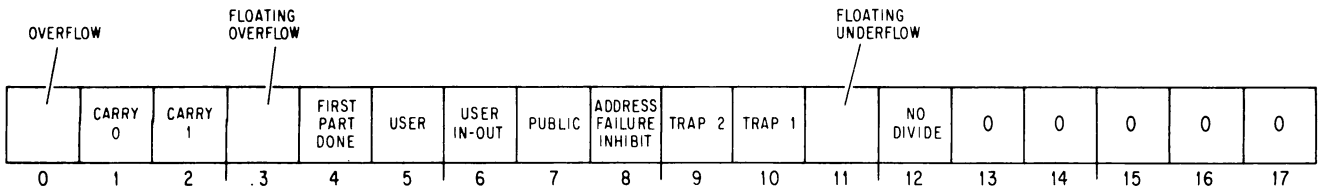
2.9 PROGRAM CONTROL

The program control class of instructions includes the unimplemented user operations [*discussed in the next section*] and the arithmetic and logical test instructions. Some instructions in this class are no-ops, as are a few of the instructions for performing logical operations. The most commonly used no-op is JFCL, which is discussed below. No-ops among the instructions previously discussed are SETA, SETAI, SETMM, CAI, CAM, JUMP, TRN, TLN, TDN, TSN. Of these, SETA, SETAI, CAI, JUMP, TRN and TLN do not use the calculated effective address to reference memory.

KA10 instruction codes 247 and 257 are reserved for instructions installed specially for a particular system. They execute as no-ops when run on a KA10 that contains no special hardware for them, but for program compatibility it is advised that they not be used regularly as no-ops.

The present section treats all program control instructions other than those mentioned above and in-out instructions that test input conditions [§2.12]. All but one of these are jumps, although the exception causes the processor to execute an instruction at an arbitrary location and may therefore be regarded as a jump with an immediate and automatic return. Also, all but two of the jumps are unconditional; one exception tests various flags, the other tests an accumulator.

Several of the jump instructions save the current contents of the program counter PC in the right half of an accumulator or memory location and save the states of various flags in the left half. The bits saved in the left half of this PC word in KI10 user mode are as shown here. In the KA10, bits 7–10 are not used. In KI10 executive mode, bit 6 receives the same flag although it has a different meaning, and bit 0 receives a different flag altogether [*see below*]. In either processor all unused bit positions are cleared.



Note that nothing is stored in bits 13–17, so when the PC word is addressed indirectly it can produce neither indexing nor further indirect addressing.

The following lists the left PC-word bit positions that receive information and explains the meaning of the flags at the time they are saved. Certain

instructions can set up these flags to restore them to their original states following an interruption or to control specific situations. The explanations assume the flags reflect normal circumstances – not arbitrary rigging. In the following an *X* in a mnemonic indicates any letter (or none) that may appear in the given position to specify the mode, *eg* ADD*X* comprises ADD, ADDI, ADDM, ADDB.

Bit — *Meaning of a 1 in the Bit*

0 Overflow – any of the following has occurred:

A single instruction has set one of the carry flags (bits 1 and 2) without setting the other.

An ASH or ASHC has left shifted a 1 out of bit 1 in a positive number or a 0 out in a negative number.

An MUL*X* has multiplied -2^{35} by itself (product 2^{70}).

A DMUL has multiplied -2^{70} by itself (product 2^{140}).

An IMUL*X* has multiplied two numbers with product $\geq 2^{35}$ or $<-2^{35}$.

An FIX or FIXR has fetched an operand with exponent <35.

Floating Overflow has been set (bit 3).

No Divide has been set (bit 12).

In user mode, bit 0 reflects the state of Overflow. But when the flags are saved in KI10 and KL10 executive mode, bit 0 represents the Disable Bypass flag, which the Monitor uses to control certain aspects of the execution of an instruction by an executive XCT [*see below and* §2.15]. Although these are two separate flags that are read in different circumstances, when a PC word is used to restore or set up the flags, bit 0 conditions both of them.

1 Carry 0 – if set with Carry 1 (bit 2) being set, causes Overflow to be set and indicates that one of the following has occurred:

An ADD*X* has added two negative numbers with sum $<-2^{35}$.

A DADD has added two negative numbers with sum less than -2^{70}.

An SUB*X* has subtracted a positive number from a negative number with difference $<-2^{35}$.

A DSUB has subtracted a positive number froma negative number with a difference $<-2^{35}$

An SOJ*X* or SOS*X* has decremented -2^{35}.

But if set with Carry 1, indicates that one of these nonoverflow events has occurred:

In an ADD*X* or DADD both summands were negative, or their signs differed and their magnitudes were equal or the positive one was the greater in magnitude.

In an SUB*X* or DSUB the signs of the operands were the same and AC was the greater or the two were equal, or the signs of the operands differed and AC was negative.

An AOJ*X* or AOS*X* has incremented –1.

An SOJ*X* or SOS*X* has decremented a nonzero number other than -2^{35}.

An MOVN*X* or DMOVN*X* has negated zero.

Remember [§2.5], overflow is determined directly from the carries, not from the flags. The carry flags give meaningful information only if no more than one instruction that can set them occurs between clearing and reading them.

2 Carry 1 – if set with Carry 0 (bit 1) being set, causes Overflow to be set and indicates that one of the following has occurred:

An ADD*X* has added two positive numbers with sum $\geqslant 2^{35}$.

A DADD has added two positive numbers with sum 2^{70}.

An SUB*X* has subtracted a negative number from a positive number with difference $\geqslant 2^{25}$.

A DSUB has subtracted a negative number from a positive number with difference $\geqslant 2^{70}$.

An AOJ*X* or AOS*X* has incremented $2^{35} - 1$.

An MOVN*X* or MOVM*X* has negated -2^{35}.

A DMOVN*X* (KL10 only) has negated -2^{70}

But if set with Carry 0, indicates that one of the nonoverflow events listed under Carry 0 has occurred.

3 Floating Overflow – any of the following has set Overflow:

Floating point instructions that cannot overflow are FLTR, DMOVN, DMOVNM and DFN.

In a floating point instruction, the exponent of the result was > 127.

DMOVNM or DFN, the exponent of the result was > 127.

Floating Underflow (bit 11) has been set.

No Divide (bit 12) has been set in an FDV*X*, FDVR*X* or DFDV.

4 First Part Done – the processor is responding to a priority interrupt between the parts of a two-part instruction or to a page failure in the second part. A 1 in this bit indicates that the first part has been completed, and this fact should be taken into account when the processor restarts the instruction at the beginning upon the return to the interrupted program. *Eg* if an ILDB or IDPB is interrupted after the processing of the pointer but before the processing of the byte, the pointer now points not to the last byte, but rather to the byte that should be handled at the return [§2.13]. Thus when the processor restarts the instruction, it must retrieve the pointer but *not* increment it.

Although this flag is set upon completion of the first part of every interruptable two-part instruction, it is seldom relevant to the programmer as it is always cleared by the completion of the second part. The flag is seen only in an interruption, and its effect on the repeated first part is automatic provided only that it is properly restored at the return.

Besides indicating a priority interrupt in the middle of a byte instruction, the KI10 First Part Done indicates a page failure in the processing of a byte, in the transfer of the second (low order) word in a DMOVEM or DMOVNM, or in a noninterrupt data IO instruction that results from a block IO instruction (following the processing of the pointer [§2.12]).

5 User – the processor is in user mode [§§2.15, 2.16].

In the KA10, User In-out is applicable only to user mode [§2.16]. In the KI10 and KL10, this flag has the stated effect when the processor is in user mode, but is used in executive mode to control certain aspects of the execution of an instruction by an executive *XCT [see below and* §2.15].

6 User In-out – even with the processor in user mode, there are no instruction restrictions (but memory restrictions still apply).

7 Public (KI10 and KL10) – the last instruction performed was fetched from a public area of memory, *ie* the processor is in user mode public or executive mode supervisor.

8 Address Failure Inhibit (KI10 and KL10) – an address failure cannot occur during the next instruction [§2.15].

9 Trap 2 (KI10 and KL10) – if bit 10 is not also set, pushdown overflow has occurred. Unless the executive paging system is disabled, the setting of this flag immediately causes a trap as explained at the end of this section. At present, bits 9 and 10 cannot be set together by any hardware condition.

10 Trap 1 (KI10 and KL10) – if bit 9 is not also set, arithmetic overflow has occurred. Unless the executive paging system is disabled, the setting of this flag immediately causes a trap as explained at the end of this section. At present, bits 9 and 10 cannot be set together by any hardware condition.

11 Floating Underflow – in a floating point instruction, the exponent of the result was <–128 and Overflow and Floating Overflow have been set.

12 No Divide – any of the following has set Overflow:

In a DIV*X* the dividend was greater than or equal to the divisor.

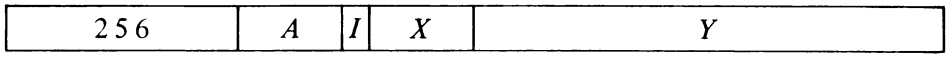
In an IDIV*X* the divisor was zero, or the dividend was -2^{35} and the divisor was ±1.

In an FDV*X* FDVR*X* or DFDV the divisor was zero, or the dividend fraction was greater than or equal to twice the divsor fraction in magnitude; in either case Floating Overflow has been set.

ADJBP the number of bytes per word was zero (see 2.3).

If normalized operands are used, only a zero divisor can cause floating division to fail.

XCT Execute



In user mode or in the KA10, execute the contents of location *E* as an instruction. Any instruction may be executed, including another XCT. If an XCT executes a skip instruction, the skip is relative to the location of the XCT (the first XCT if there are several in a chain). If an XCT executes a jump, program flow is altered as specified by the jump (no matter how many XCTs precede a jump instruction, when PC is saved it contains an address one greater than the location of the first XCT in the chain).

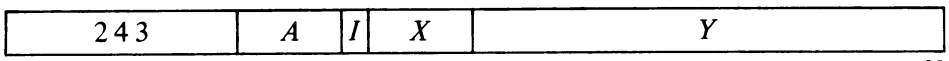
In KI10 executive mode this instruction performs as stated only when *A* is zero. Nonzero *A* results in a so called "executive XCT", whose ramifications are far more widespread than indicated here [*for details refer to* §2.15].

In user mode and in the KA10, the *A* portion of this instruction is ignored. It should then be zero for compatibility with KI10 executive mode and possible future use even in user mode.

CAUTION

In concealed or kernel mode, an XCT that executes an instruction in a public page places the processor in public or supervisor mode. Hence unless the executed instruction changes PC to a public area, the instruction following the XCT must be a valid entry point back into the concealed area or a page failure, in particular a proprietary violation, will result. A valid entry point is one containing a particular form of the JRST instruction described below.

JFFO Jump if Find First One



If AC contains zero, clear AC *A*+1 and go on to the next instruction in sequence.

If AC is not zero, count the number of leading 0s in it (0s to the left of the leftmost 1), and place the count in AC *A*+1. Take the next instruction from location *E* and continue sequential operation from there.

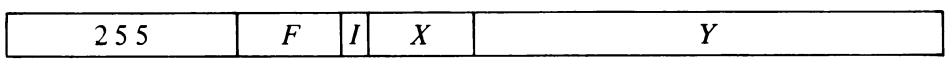
In either case AC is unaffected, the original contents of AC *A*+1 are lost.

Note that when AC is negative, the second accumulator is cleared, just as it would be if AC were zero.

To left-normalize a positive integer in AC: ▲

```
JFFO    AC,.+1
LSH     AC,-1(AC+1)
```

JFCL Jump on Flag and Clear



If any flag specified by *F* is set, clear it and take the next instruction from

location E, continuing sequential operation from there. Bits 9–12 are programmed as follows.

Bit	*Flag Selected by a 1*
9	Overflow
10	Carry 0
11	Carry 1
12	Floating Overflow

This instruction can be used simply to clear the selected flags by having the jump address point to the next consecutive location, as in

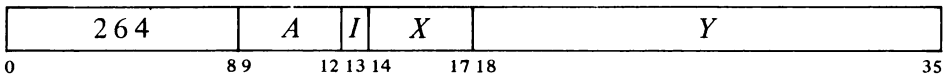
```
JFCL  17,.+1
```

which clears all four flags without disrupting the normal program sequence. A JFCL that selects no flag is the fastest no-op as it neither fetches nor stores an operand, and bits 18–35 of the instruction word can be used to store information.

To select one or a combination of these flags (which are among those described above) the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits, but MACRO recognizes mnemonics for some of the 13-bit instruction codes (bits 0–12).

JFCL	JFCL 0,	No-op	25500
JOV	JFCL 10,	Jump on Overflow	25540
JCRY0	JFCL 4,	Jump on Carry 0	25520
JCRY1	JFCL 2,	Jump on Carry 1	25510
JCRY	JFCL 6,	Jump on Carry 0 or 1	25530
JFOV	JFCL 1,	Jump on Floating Overflow	25504

JSR Jump to Subroutine

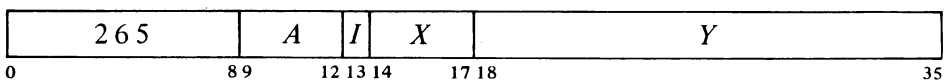


Place the current contents of the flags (as described above) in the left half of location E and the contents of PC in the right half (at this time PC contains an address one greater than the location of the JSR instruction). Take the next instruction from location $E + 1$ and continue sequential operation from there. The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared.

The A portion of this instruction is reserved for future use and should be zero (at present it is ignored).

While the processor is in user mode, if this instruction is executed as an interrupt instruction or by a KA10 MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode, clearing Public. (In the KI10 an interrupt that is not dismissed automatically returns control to kernel mode.)

JSP Jump and Save PC

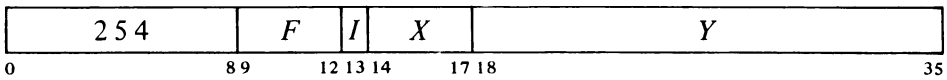


Place the current contents of the flags (as described above) in AC left and

the contents of PC in AC right (at this time PC contains an address one greater than the location of the JSP instruction). Take the next instruction from location *E* and continue sequential operation from there. The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared.

While the processor is in user mode, if this instruction is executed as an interrupt instruction or by a KA10 MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode, clearing Public. (In the KI10 an interrupt that is not dismissed automatically returns control to kernel mode.)

JRST Jump and Restore



Perform the functions specified by *F*, then take the next instruction from location *E* and continue sequential operation from there. Bits 9–12 are programmed as follows.

Bit	*Function Produced by a 1*
9	Restore the channel on which the highest priority interrupt is currently being held [§2.13]. This function cannot be performed at all in a KI10 user or supervisor program and cannot be performed in a KA10 user program unless User In-out is set. Instead of restoring the channel, it acts just like an MUUO [§2.10]. ▲
10	Halt the processor. When it stops, the AR lights on the KI10 and the MA lights on the KA10 display an address one greater than that of the location containing the instruction that caused the halt, and PC displays the jump address (the location from which the next instruction will be taken if the operator causes the processor to resume operation without changing PC). This function cannot be performed at all in a KI10 user or supervisor program and cannot be performed in a KA10 user program unless User In-out is set. Instead of halting the processor, it acts just like an MUUO [§2.10]. ▲
11	Restore the flags listed above from the left half of the word in the last location referenced in the effective address calculation. Hence to restore flags requires that the JRST instruction use indexing or indirect addressing. Restoration of all but the user and Public flags is directly according to the contents of the corresponding bits as given above: a flag is set by a 1 in the bit, cleared by a 0. A 1 in bit 5 sets User but a 0 has no effect, so the Monitor can restart a user program by restoring flags but the user cannot leave user mode by this method. A 0 in bit 6 clears User In-out, but a 1 sets it only if the JRST is being performed

AR or MA actually displays the address of the location that would have been executed next had the JRST been replaced by a no-op. So except for a JRST in a priority interrupt, the lights point to the location one beyond that containing the instruction that caused the halt. This instruction is ordinarily the JRST or perhaps an XCT, but could even be a UUO.

By manipulating the contents of the left half word used to restore the flags, the programmer can set them up in any desired way except that a user program cannot clear User or set User In-out, and

no public program can clear Public for itself. As an example, setting First Part Done prevents incrementing in the next ILDB, IDPB or noninterrupt KI10 block IO instruction provided there is no intervening JSR, JSP or PUSHJ. Note that if overflow traps are enabled, setting a trap flag immediately causes one.

by the Monitor, *ie* if User is clear. A 1 in bit 7 sets Public, but a 0 clears it only if the JRST is being performed in executive mode with a 1 in bit 5 (*ie* User is being set). These conditions imply that the processor is entering user mode: hence the user cannot enter concealed mode by clearing Public; and although the supervisor can place the processor in user mode concealed, it cannot use this procedure to enter kernel mode.

12 *KA10.* Enter User mode. The user program starts at relocated location *E.*
KI10 and *KL10.* The instruction is simply a jump except when fetched from a nonpublic area, in which case it clears Public. In other words a location containing a JRST 1, is a valid entry to a nonpublic area and the instruction places the processor in concealed or kernel mode.

Combinations other than the six listed will trap as an MUUO on the KL10.

To produce one or a combination of these functions the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits, but MACRO recognizes mnemonics for the most important 13-bit instruction codes (bits 0–12).

JRST	JRST 0,	Jump	25400
JRST10,	JRST 10,	Jump and Restore Interrupt Channel	25440
HALT	JRST 4,	Halt	25420
JRSTF	JRST 2,	Jump and Restore Flags	25410
PORTAL	JRST 1,	Allow Nonpublic Entry (KI10) Jump to User Program (KA10)	25404
JEN	JRST 12,	Jump and Enable	25450

JEN completes an interrupt by restoring the channel and restoring the flags for the interrupted program.

In a JRSTF or JEN the flags are restored from bits 0–12 of the final word retrieved in the effective address calculation; hence any JRST with a 1 in bit 11 must use indirect addressing or indexing, which takes extra time. If the PC word was stored in AC (as in a JSP), a common procedure is to use AC to index a zero address (*eg*, JRSTF (AC)), so its right half becomes the effective (jump) address. If the PC word was stored in core (as in a JSR), one must address it indirectly (remember, bits 13–17 of the PC word are clear, so again its right half is the effective address). A JRSTF (AC) is considerably faster than a JRSTF @PCWORD.

CAUTION

Giving a JRSTF or JEN without indexing or indirect addressing restores the flags from the instruction code itself.

While the KA10 is in user mode, if this instruction is executed as an interrupt instruction or by an MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode.

The following table shows the situations in which each of the 16 combinations of JRST is legal on KA10, KI10, and KL10.

		Executive Kernel*	Executive Supervisor*	User	User IOT
JRST	0	AIL	AIL	AIL	AIL
PORTAL**	1	AIL	AIL	AIL	AIL
JRST F	2	AIL	AIL	AIL	AIL
	3	AI–	AI–	AI–	AI–
HALT	4	AIL	A––	–––	AIL
	5	AI–	A––	–––	AI–
	6	AI–	A––	–––	AI–
	7	AI–	A––	–––	AI–
	10	AIL	A––	–––	A–L
	11	AI–	A––	–––	A––
JEN	12	AIL	A––	–––	AIL
	13	AI–	A––	–––	A––
	14	AI–	A––	–––	AI–
	15	AI–	A––	–––	AI–
	16	AI–	A––	–––	AI–
	17	AI–	A––	–––	AI–

*In the KA10, exec mode is not subdivided into kernel and supervisor modes

**Allow Nonpublic Entry (KI10), Jump to User Program (KA10)

LEGAL IN
A = KA10
I = KI10
L = KL10

JFCL is the only jump that can test any of the flags directly. In fact it is the only basic program control instruction that can do so – several of the flags can be tested as processor conditions by in-out instructions, but these are ordinarily illegal in user programs anyway. But JFCL can test only four of the flags, and it saves no information for a subsequent return from a subroutine. Hence it serves as a branch point for entry into either one of two main paths, which may or may not have a later point in common. *Eg*, it may test the carry flags simply to take appropriate action in a double precision fixed point routine.

JSR and JSP are regularly used to call subroutines. They are unconditional, but the execution of such an instruction can be the result of a decision made by any conditional skip or jump. In the case of the flags, a basic overflow test and subroutine call can be made as follows.

```
JOV     .+2
JRST    .+2         ;Faster than skipping
JSR     OVRFLO      ;Jump over this if Overflow clear
.
.
.
```

The fastest skip is CAIA in the KA10, TRNA in the KI10.

If we wish to go to the DIVERR routine when No Divide is set, we must first read the flags into a test accumulator T and then use a test instruction.

```
JSP     T,.+1       ;Store flags but continue in sequence
TLNE    T,40        ;40 left selects bit 12
JSR     DIVERR      ;Skip this if No Divide clear
.
.
.
```

A subroutine called by a JSR must have its entry point reserved for the PC word. Hence it is nonreentrant: the JSR modifies memory so the subroutine cannot be shared with other programs. The JSP requires an accumulator, but it is faster and is convenient for argument passing. To return from a JSR-called subroutine one usually addresses the PC word indirectly, returning to the location following the JSR. But there are two ways to get back from a JSP. We can address the PC word indirectly with a JRST @AC (or JRSTF @AC if the flags must be restored), but we can also get it by addressing AC as an index register: JRST (AC). By using the second return method we can place *N* words of data for the subroutine immediately after the call, and return to the location following the data by giving a JRST *N*(AC).

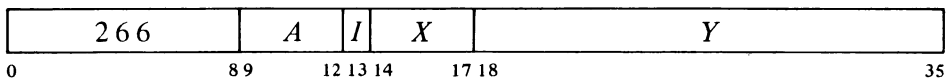
Suppose we wish to call a print subroutine and supply the words from which the characters are to be taken. Our main program would contain the following:

```
JSP     T,PRINT     ;Put PC word in accumulator T
.                   ;Text inserted here by ASCIZ pseudo-
.                   ;instruction, which automatically
                    ;places a zero (null) character at the
                    ;end
...                 ;Next instruction here
```

The subroutine can use T as a byte pointer which already addresses the first word of data. For the print routine, characters are loaded into another accumulator CH.

```
PRINT:  HRLI    T,440700    ;Initialize left half of pointer
        ILDB    CH,T        ;Increment pointer and load byte
        JUMPE   CH,1(T)     ;Upon reaching zero character return
                            ;to one beyond last data word
          .                 ;Print routine
          .
          .
        JRST    PRINT+1     ;Get next character
```

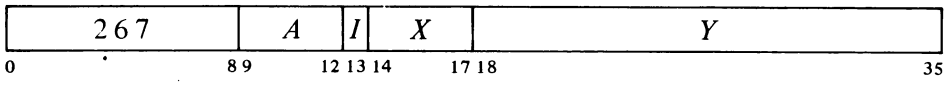
JSA Jump and Save AC



Place AC in location E, the effective address E in AC left, and the contents of PC in AC right (at this time PC contains an address one greater than the location of the JSA instruction). Take the next instruction from location $E+1$ and continue sequential operation from there. The original contents of E are lost.

While the KA10 is in user mode, if this instruction is executed as an interrupt instruction or by an MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode.

JRA Jump and Restore AC



Place the contents of the location addressed by AC left into AC. Take the next instruction from location E and continue sequential operation from there.

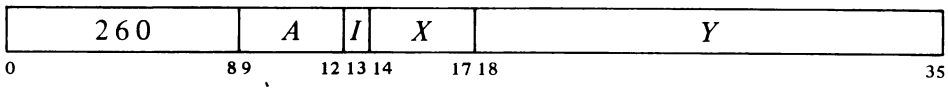
A JSA combines advantages of the JSR and JSP. JSA does modify memory, but it saves PC in an accumulator without losing its previous contents (at a cost of not saving the flags). It is thus convenient for multiple-entry subroutines. In a subroutine called by a JSR, the returning JRST must refer to the (single) entry point. Since a JRA can retrieve the original PC by addressing AC as an index register, it is independent of any entry point without tying up an accumulator to the extent a JSP would.

The accumulator contents saved by a JSA are restored by a JRA paired with it despite intervening JSA-JRA pairs. Hence these instructions are especially useful for nesting subroutines, as shown by this example.

```
                .                       ;Main program
                .
                .
                JSA     17,S1           ;Call to first subroutine (A)
                .
                .
S1:             0                       ;First subroutine starts here
                .
                .
                JSA     17,S2           ;Call to second subroutine (B)
                .
                .
                JRA     17,(17)         ;Return to A + 1 in main program
S2:             0                       ;Second subroutine starts here
                .
                .
                JSA     17,S3           ;Call to third subroutine (C)
                .
                .
                JRA     17,(17)         ;Return to B + 1 in first subroutine
S3:             0                       ;Third subroutine starts here
                .
                .
                JRA     17,(17)         ;Return to C + 1 in second subroutine
```

To call the next deeper subroutine at any level, a JSA places *E* and PC in the left and right of AC 17, saves the previous contents of AC 17 in *E* (the first subroutine location), and jumps to *E* + 1. To return to the next higher level, a JRA restores the previous contents of AC 17 from the location addressed by AC 17 left (the first subroutine location) and jumps to the location addressed by AC 17 right (the location following the JSA in the higher subroutine). If *N* lines of data for the next subroutine follow a JSA, the return to the location following the data is made by giving a JRA 17,*N*(17).

PUSHJ Push Down and Jump



Add one to each half of AC and place the result back in AC. If the addition causes the count in AC left to reach zero, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10 and KL10. Then place the current contents of the flags (as described above) in the left half of the location now addressed by AC right and the contents of PC in the right half of that location (at this time PC contains an address one greater than the location of the PUSHJ instruction). Take the next instruction from location E and continue sequential operation from there.

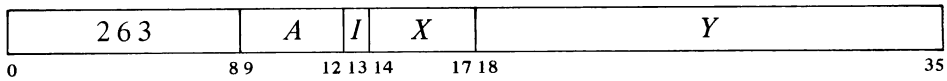
In the KI10 and KL10, a PUSHJ executed as an interrupt instruction cannot set Trap 2.

The flags are unaffected except First Part Done, Address Failure Inhibit, and the trap flags, which are cleared. However, pushdown overflow overrides the Trap 2 clear, so if the list overflows, Trap 2 sets and the KI10 traps instead of jumping. The original contents of the location added to the list are lost.

Note: The KA10 increments the two halves of AC by adding $1\,000001_8$ to the entire register. In the KI10 the two halves are handled independently.

While the processor is in user mode, if this instruction is executed as an interrupt instruction or by a KA10 MUUO, bit 5 of the PC word stored is 1 and the processor leaves user mode, clearing Public. (In the KI10 an interrupt that is not dismissed automatically returns control to kernel mode.)

POPJ Pop Up and Jump



Subtract one from each half of AC and place the result back in AC. If the subtraction causes the count in AC left to reach −1, set the Pushdown Overflow flag in the KA10, set the Trap 2 flag in the KI10. Take the next instruction from the location addressed by the right half of the location that was addressed by AC right *prior* to the decrementing, and continue sequential operation from there.

I, *X* and *Y* are reserved for future use and should be zero (at present *E* is ignored). In the KI10 a POPJ executed as an interrupt instruction cannot set Trap 2.

Note: The KA10 decrements the two halves of AC by subtracting $1\,000001_8$ from the entire register. In the KI10 the two halves are handled independently.

CAUTION

The jump is completed before the processor responds to overflow, whether by trap or interrupt. Hence it is impossible to determine the location of the POPJ that caused the overflow.

The address of the top item in the pushdown list is kept in the right half of the pointer in AC, and the program can keep a control count in the left half. In the KA10, incrementing and decrementing both halves of AC together is effected by adding and subtracting $1\,000001_8$. Hence a count of −2 in AC left is increased to zero if $2^{18} - 1$ is incremented in AC right, and conversely, 1 in AC left is decreased to −1 if zero is decremented in AC right.

Since the pushdown list is independent of the subroutine called, PUSHJ-POPJ can be used like JSA-JRA for multiple entries. Moreover, ordering by level is inherent in the structure of a pushdown list [§2.2], so paired PUSHJ-POPJ instructions are excellent for nesting subroutines: there can be any number of subroutines at any level, each with more subroutines nested within it. Recursive subroutines are also possible.

Unlike JSA-JRA, the pushdown instructions tie up an accumulator, but the usual procedure is to keep both data and jump addresses in a single list so only one AC is required for the most complex pushdown operations. The programmer must keep track of whether a given entry in the list is data or a PC word; in other words, every item inserted by a PUSH should be removed by a POP, and every PUSHJ should be matched by a POPJ. If flag restoration is desired, the returning

```
        POPJ    P,
```

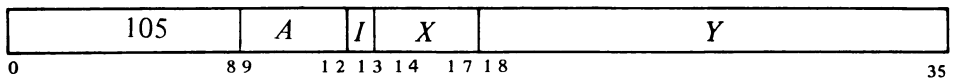
can be replaced by

```
        POP     P,AC
        JRSTF   (AC)
```

which requires another accumulator. If the flags are not important, data may be stored in the left halves of the PC words in the stack, reducing the required pushdown depth.

By trapping or checking overflow and keeping a control count in AC left, the programmer can set a limit to the size of the list by starting the count negative, or he can prevent the program from extracting more items than there are in the list by starting the count at zero, but he cannot do both at once. If only jump addresses are kept in the list, the first procedure limits the depth of nesting. A technique to catch extra POPJs is to put a PC word addressing an error routine at the bottom of the list.

ADJSP **Adjust Stack Pointer** (KL10 Only)



Add *E* to each half of AC and place the result back in AC. If a negative adjustment changes the count in AC_L from positive to negative, or a positive adjustment changes the count in AC_L from negative to positive, set Trap 2 (pushdown overflow) as shown in the chart below.

AC_L			
Original	*Result*	*E*	*Overflow*
Positive	Positive	x	No
Negative	Negative	x	No
Positive	Negative	Positive	No
Negative	Postive	Negative	No
Positive	Negative	Negative	Yes
Negative	Positive	Positive	Yes

NOTE

The KL10 adds *E* to each half of AC separately with no carries between halves.

Overflow Trapping

In the performance of a program there are many events that cannot be foreseen and whose occurrence requires special action by the program. There are instructions that test for the conditions produced by such events, but in say a long string of computations, it would be both cumbersome and time consuming to test for overflow at every step. It is far better simply to allow an event such as overflow to break right into the normal program sequence.

NOTE

This feature is not available in the KA10. That processor is limited to the use of internal conditions that can act through the priority interrupt system [§2.14].

For situations of this nature, various internal conditions can act through the priority interrupt system. However the processor also has a trapping mechanism that allows conditions due directly to the program, and which are often permitted to happen as a matter of course, to interrupt the program sequence without recourse to the interrupt system. In some cases, traps are used to handle the restrictions that play a role in program and memory management [*as explained in later sections*], but here we are concerned specifically with action by the processor in response to overflow.

Overflow produced by an interrupt instruction cannot be detected. In any other circumstances, an instruction in which an arithmetic overflow condition occurs sets Overflow and Trap 1, and an instruction in which a pushdown overflow occurs sets Trap 2. At the completion of an instruction in which either trap flag is set, rather than going on to the next instruction as specified by PC, the processor instead executes an instruction taken from a particular location in the process table for the program (user or executive). The location as a function of the trap flags set is as follows.

Note that it is the overflow condition that sets Trap 1 – not the state of the Overflow flag. Hence an overflow is trapped even if Overflow is already set.

Note also that the trap flags have no effect at all when executive paging is disabled [§2.15].

Trap Flags Set	*Trap Type*	*Trap Number*	*Location*
Trap 1 only	Arithmetic overflow	1	421
Trap 2 only	Pushdown overflow	2	422
Trap 1 and 2	Not used by hardware	3	423

A trap can be produced artificially simply by setting up the trap flags with a JRSTF or MUUO. In this way the program can also use trap number 3, which at present cannot result from any hardware-detected condition (it is reserved for future use by DEC).

A trap instruction is executed in the same address space as the instruction that caused it. Overflow in a user instruction traps to a location in the user process table, and any addresses used in the instruction in that location are interpreted in the user address space. Thus a user program can handle its own traps, *eg* by requesting the Monitor to place a PUSHJ to a user routine in the trap location. An MUUO must be used if the Monitor is to handle a user-caused trap.

The location of the instruction that caused the overflow can be determined from PC unless the instruction jumped, in which case its location can be determined only for a PUSHJ, from the stack entry.

The trap instruction (the final instruction in an XCT and/or LUUO string) clears the trap flags, so the processor returns to the interrupted program unless the trap instruction changes PC. Thus the trap instruction can be a no-op (which ignores the trap), a skip, a jump, or anything else. However, should the trap instruction itself set a trap flag (not necessarily the same one), a second trap occurs.

An arithmetic instruction that overflows on every iteration produces an infinite loop if used as a trap instruction for arithmetic overflow. A pushdown instruction in a pushdown overflow trap can overflow only once. (The memory allocated to a pushdown stack should have at least one extra location to handle this case – two extras if the program and the trap both use the same pointer.)

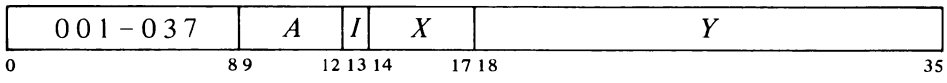
An interrupt can occur between an instruction that overflows and the trap instruction, but the latter will be performed correctly upon the return provided the interrupt is dismissed automatically or the interrupt routine restores the flags properly. If a single instruction causes both overflow and a page failure, the latter has preference; but the overflow trap will be taken care of after the offending instruction has been restarted and completed successfully. A trap instruction that causes a page failure does not clear the trap flags; hence after the page failure is taken care of, the trap instruction will correctly handle the trap when it is restarted.

2.10 UNIMPLEMENTED OPERATIONS

Codes not assigned as specific instructions act as unimplemented operations, wherein the word given as an instruction is trapped and must be interpreted by a routine included for this purpose by the programmer. Codes in the range 001–077 are unimplemented user operations, or UUOs. Half of these (001–037) are for the local use of the user or Monitor (LUUOs); the other half (040–077) are set aside for user communication with the Monitor (MUUOs) and are interpreted by it (although they may be used by the Monitor as well). Codes 100 and above that are not used for instructions are regarded as the "unassigned codes"; 000 is not regarded as a legal code at all. Instructions that violate the instruction restrictions act in the same manner as MUUOs.

These are convenience mnemonics that mean nothing to the assembler. UUOs are also sometimes called "programmed operators".

Local Unimplemented User Operation



Store the instruction code, *A* and the effective address *E* in bits 0–8, 9–12 and 18–35 respectively of location 40; clear bits 13–17. Execute the instruction contained in location 41. The original contents of location 40 are lost.

Every LUUO uses some pair of locations numbered 40 and 41, but which such pair depends upon the circumstances. An LUUO in a user program uses relocated locations 40 and 41 and is thus entirely a part of and under control of the user program. An LUUO in KA10 executive mode uses unrelocated locations. In KI10 executive mode an LUUO uses locations 40 and 41 in the executive process table.

If a single memory serves as memory number 0 for two KA10 processors, the second (with the trap offset) uses unrelocated 140-141 and 160-161 respectively for each instance in which 40-41 and 60-61 are given here. The offset does not apply to user LUUOs as it is assumed the Monitor would relocate these to different physical blocks.

The actions of MUUOs and unassigned codes depend to a considerable degree on the processor. All use at least two consecutive locations, where the first receives the information specified above for an LUUO (in the KI10 a third nonconsecutive location is also used). The unassigned codes are included so that the Monitor steps in when a user gives an incorrect code. The code 000 acts in exactly the same way as an MUUO but is not a standard communication code: it is included so that control returns to the Monitor should a user program wipe itself out.

KL10 and KI10. MUUOs and unassigned codes in user or executive mode act in exactly the same way. They store the information specified above for an LUUO in location 424 of the user process table, save the flags and PC (the current PC word) in location 425, in the KL10, save the process context word in location 426, set up the flags and PC according to a new PC word taken from an additional location, and restart the processor in normal sequence at the location then addressed by PC. In the PC word saved in location 425, bit 0 may represent either Overflow or Disable Bypass depending upon the mode the processor is in when the MUUO is given. If the MUUO is given directly by the program, the address in the right half of the PC word saved is one greater than the location of the MUUO; otherwise it depends upon the circumstances in which the MUUO is executed. The new PC word can be taken from among the eight locations in the user process table listed here depending upon the mode at the time the MUUO is given, and whether or not it is executed as the result of a trap (page failure on the KI10 or overflow on either processor).

The unassigned codes on KI10 are 100-107, 114-117, 123 and 247.

The unassigned codes on KL10 are 100-104, 106, and 247.

Note that even in a dedicated system, the program must still define a user process table.

Mode	*Execution*	*Location*
Kernel	No trap	430
Kernel	Trap	431
Supervisor	No trap	432
Supervisor	Trap	433
Concealed	No trap	434
Concealed	Trap	435
Public	No trap	436
Public	Trap	437

There are no restrictions on the manner in which the new PC word of an MUUO can set up the flags. It can switch the processor from any mode to any other. A 1 in bit 0 sets both Overflow and Disable Bypass; a 0 clears both. Hence bit 0 should be adjusted to produce the desired state in the flag that is relevant to the mode the processor is entering.

Note that unless executive paging is disabled, setting a trap flag immediately causes a trap.

Note that in executive mode, LUUOs and MUUOs act identically.

Codes 247 and 257, although not assigned as specific instructions, are nonetheless not regarded as "unassigned" codes. They execute as no-ops unless implemented by special hardware.

KA10. MUUOs and unassigned codes, regardless of mode, perform exactly the operations given above for an LUUO with the exception that MUUOs use unrelocated 40-41 and unassigned codes use unrelocated 60-61 (140-141 and 160-161 for a second processor). The unassigned codes are 100–127. The codes 130–177, which are the floating point and byte manipulation instructions, are equivalent to the unassigned codes if unimplemented, *ie* if the hardware for them is not included. In this case all codes 100–177 trap to unrelocated 60-61.

The important point is that an MUUO or unassigned code results in executing an instruction in an unrelocated location, *ie* an instruction under the control of the Monitor. This would most likely be a jump that leaves user mode, saves the PC word and enters a routine to interpret the MUUO configuration. In the instruction descriptions, any reference to events resulting from execution by an MUUO should be taken to include the unassigned and illegal codes as well.

2.11 PROGRAMMING EXAMPLES

Before continuing to input-output and related subjects, let us consider some simple programs that demonstrate the use of a variety of the instructions described thus far.

Processor Identification

The instruction repertoires of the KI10, the KA10, and the 166 processor used in the PDP–6 are very similar, and most programs require no changes to run on any of them. Because of minor differences and the fact that some instructions are not available on the earlier machines, a program that is to be compatible with all three should have some way of distinguishing which machine it is running on. This simple test suffices.

```
JFCL    17,.+1      ;Clear flags
JRST    .+1         ;Change PC
JFCL    1,PDP6      ;PDP–6 has PC Change flag
MOVNI   AC,1        ;Others do not, make AC all 1s
AOBJN   AC,.+1      ;Increment both halves
JUMPN   AC,KA10     ;KA10 if AC=1000000
BLT     AC,0
JUMPN   AC,KL10     ;KL10 if AC=1,,1
JRST    KI10        ;KI10 if AC=0 (no carry between
                    ;halves)
```

Parity

Parity procedures are used regularly to check the accuracy of stored information. Parity generation and checking is generally handled automatically by memory and high speed, block-oriented peripheral devices, but must be handled by the program for character-oriented devices. Consider 8-bit characters, for which the program carries out two procedures: for output it

generates a parity bit from seven data bits to produce an 8-bit character with parity; following input it checks the parity of the eight bits received. In either case however, the program can simply find the parity of an 8-bit character, by regarding the seven output data bits as eight including an irrelevant extra bit. The two procedures then differ only in the final action. In the first case the program uses the result to adjust the eighth bit for correct parity, whereas in the second it checks the result for an indication of error.

Assuming the character is right-justified in accumulator A, the simplest and quickest procedure would be to use A to index an XCT into a table, each of whose locations contains an instruction that adjusts the parity for output or jumps to a routine for erroneous input. This procedure would normally be unacceptable because of the very large memory requirements. However the table can be reduced to sixteen entries without excessive loss in speed, by exclusive oring the left and right halves of the character and indexing on the result (parity is invariant under the exclusive OR function, which essentially disposes of pairs of 1s). This example, which uses a second accumulator T for character manipulation, requires six memory references to generate odd parity.

We assume the rest of A, outside the character, is clear, as it would be were the character placed in A by a load-byte instruction or a DATAI. The next two examples, however, work even if the rest of A is not clear.

```
PARITY: MOVEI  T,(A)       ;Copy character in T
        LSH    T,-4        ;Line up halves
        XORI   T,(A)       ;Reduce paritywise to 4 bits
        ANDI   T,17        ;Wipe out unwanted bits
        XCT    PARTAB(T)   ;Execute indicated table item
        POPJ   P,

PARTAB: XORI   A,200       ;0 - change high bit
        JFCL               ;1 - no-op
        JFCL               ;2
        XORI   A,200       ;3
        JFCL               ;4
        XORI   A,200       ;5
        XORI   A,200       ;6
        JFCL               ;7
        JFCL               ;10
        XORI   A,200       ;11
        XORI   A,200       ;12
        JFCL               ;13
        XORI   A,200       ;14
        JFCL               ;15
        JFCL               ;16
        XORI   A,200       ;17
```

Numbers of memory references and locations given do not include those for the POPJ, which we will regard as subroutine overhead. Similarly every example also requires that the program give a PUSHJ to get to the subroutine.

To handle even parity, interchange the JFCLs and XORIs in the table, or change the MOVEI T,(A) to MOVEI T,200(A).

The next example does exactly the same thing but substitutes a little more computation for use of a table. In other words it takes a little more time (7½ memory references average) but less than half the memory.

```
PARITY:  MOVEI  T,200(A)    ;Copy character with high bit comple-
         LSH    T,-4        ;mented, then fold copy into 4 bits
         XORI   T,(A)       ;with opposite parity
         TRCE   T,14        ;Are left two both 0?
         TRNN   T,14        ;Or both 1?
         XORI   A,200       ;Yes, change high bit
         TRCE   T,3         ;Are right two both 0?
         TRNN   T,3         ;Or both 1?
         XORI   A,200       ;Yes, change for even, restore for odd
         POPJ   P,
```

For even parity change the address in the MOVEI from 200 to 0.

Finally let us consider the extreme of substituting computation for memory. Starting with the character *abcdefgh* right-justified in A, we first copy it in T and then duplicate it twice to the left producing

abc def gha bcd efg hab cde fgh

where the bits (in positions 12–35) are grouped corresponding to the octal digits in the word. Anding this with

001 001 001 001 001 001 001 001

retains only the least significant bit in each 3-bit set, so we can represent the result by

cfadgbeh

where each letter represents an octal digit having the same value (0 or 1) as the bit originally represented by the same letter. Multiplying this by 11111111_8 generates the following partial products:

```
                c f a d g b e h
              c f a d g b e h
            c f a d g b e h
          c f a d g b e h
        c f a d g b e h
      c f a d g b e h
    c f a d g b e h
  c f a d g b e h
```

Since any digit is at most 1, there can be no carry out of any column with fewer than eight digits unless there is a carry into it. Hence the octal digit produced by summing the center column (the one containing all the bits of the character) is even or odd as the sum of the bits is even or odd. Thus its least significant bit (bit 14 of the low order word in the product) is the parity of the character, 0 if even, 1 if odd.

The above may seem a very complicated procedure to do something trivial, but it is effected by this quite simple sequence:

```
PARITY:  MOVEI  T,(A)       ;Copy in T
         IMULI  T,200401    ;Duplicate twice
         AND    T,ONES      ;Pick LSBs
```

```
        IMUL    T,ONES      ;Generate product
        TLNN    T,10        ;Is bit 14 odd?
        XORI    A,200       ;No, change parity
        POPJ    P,
        .
        .
        .
ONES:   11111111
```

This procedure uses a minimum of both memory references and memory space, but takes considerably more time because the instructions themselves are slow.

The following table shows the trade-off of memory references against memory space for the above four procedures. The time is proportional to the number of references except in case 4.

	References	*Locations*
1.	2	257
2.	6	21
3.	7½	9
4.	7½	7

Counting Ones

Suppose we wish to count the number of 1s in a word. We could of course check every bit in the word. But there is a quicker way if we remember that in any word and its twos complement the rightmost 1 is in the same position, both words are all 0s to the right of this 1, and no corresponding bits are the same to the left (the parts of both words at the left of the rightmost 1 are complements). Hence using the negative of a word as a mask for the word in a test instruction selects only the rightmost 1 for modification. The example uses three accumulators: the word being tested (which is lost) is in T, the count is kept in CNT, and the mask created in each step is stored in TEMP.

```
        MOVEI   CNT,0       ;Clear CNT
        MOVN    TEMP,T      ;Make mask to select rightmost 1
        TDZE    T,TEMP      ;Clear rightmost 1 in T
        AOJA    CNT,.-2     ;Increase count and jump back
        ...                 ;Skip to here if no 1s left in T
```

CNT is increased by one every time a 1 is deleted from T. After all 1s have been removed, the TDZE skips.

The preceding example uses little memory, but contains a loop so the time it takes is proportional to the number of 1s. The next example takes more memory but is constant; hence it is slower than the above when there are few 1s (less than eight), but is much faster when there are many. The word, which is lost, is in accumulator A, and the answer appears in accumulator

A+1 (for convenience we let B = A+1). The routine (due to Gosper, Mann and Leonard*) has three distinct parts and is based on the fact that in a binary word, counting 1s is equivalent to calculating the sum of the digits. The first part, of seven instructions, manipulates the *octal* digits of the word so as to replace each digit by the number of 1s in it. Taking D as an octal digit and $[x]$ as the largest integer contained in x, the algorithm used to make the substitution is

$$D - [D/2] - [D/4]$$

Of course the computer always acts in binary terms regardless of programmer interpretation. In this case the procedure carried out on each 3-bit piece *abc* is

$$abc - ab - a$$

The instructions effect this algorithm by shifting a copy of the word right one place, masking out the LSB of each shifted octal digit to prevent it from interfering with the next digit at the right (*ie* to isolate the digits), and subtracting the shifted word from the original. The same process is then repeated, this time masking out what was originally the middle bit in each digit. That this algorithm gives the correct substitution is evident from the following table, in which it is seen that the bottom number in a given column is the sum of the bits in the octal digit given at the top of the column.

Original digit	0	1	2	3	4	5	6	7
Subtract	0	0	1	1	2	2	3	3
	0	1	1	2	2	3	3	4
Subtract	0	0	0	0	1	1	1	1
Number of 1s	0	1	1	2	1	2	2	3

We have now replaced the original word with a set of twelve numbers, whose sum is equal to the number of 1s in the original. The next three instructions add together pairs of adjacent numbers so as to replace the twelve by six whose sum is still the same. Since these new numbers are isolated in 6-bit pieces of the word, we shall revise our point of view, and regard them as digits in a number in base 64. Now any number is simply the sum of the values of its digits, *ie* of its digits each multiplied by an appropriate power of the base. Dividing each such summand by 1 less than the base gives the digit itself as remainder. Hence the third part of the routine just divides our 6-digit number by 63, producing in B a remainder that is the sum of the remainders from the individual digits, *ie* that is the sum of the digits.

```
MOVE    B,A             ;Copy in B
LSH     B,-1            ;Right one
AND     B,[333333,,333333]      ;Masks out LSBs
SUB     A,B             ;D - [D/2]
LSH     B,-1            ;Right one again
AND     B,[333333,,333333]      ;Mask out middle bits
SUBB    A,B             ;D - [D/2] - [D/4]; two copies
```

*HAKMEM 140, item 169, page 79 (*Artificial Intelligence Memorandum, No. 239,* February 29, 1972, MIT Artificial Intelligence Laboratory).

In general terms this is the statement that the sum S of the digits in any number N in base b is $N \bmod (b-1)$ – *provided* b is deliberately chosen such that $S < b - 1$. The condition holds here of course as the number of 1s in a PDP-10 word is at most 36. And it is in fact to make this

```
LSH     B,-3            ;Shift right one octal digit
ADD     A,B             ;Add numbers in digit pairs
AND     A,[070707,,070707]      ;Throw out extra pair sums

IDIVI   A,77            ;Divide by 63, sum in B
```

condition hold that the routine converts from base 8 to base 64.

If it is known that the 1s in the word are entirely contained within bits 22–35 (the right fourteen bits), we can use the following somewhat shorter routine, which is faster than the loop for more than seven 1s. It first treats the number in quaternary, replacing each digit with the number of 1s in it, and then converts from quaternary to hexadecimal.

```
MOVEI   B,(A)
LSH     B,-1
ANDI    B,12525         ;Mask out LSBs
SUBB    A,B             ;D - D/2] ; two copies

LSH     B,-2            ;Right one quaternary digit
ANDI    A,31463         ;Mask out some of digits in A
ANDI    B,31463         ;The rest in B
ADDI    A,(B)           ;Now combine digit pairs

IDIVI   A,17            ;Divide by 15, sum in B
```

Note that here we must get rid of one out of each set of two identical bit pairs before adding. This is because there can be digit overflow, *ie* a resulting hexadecimal digit can have more than two significant bits.

Number Conversion

In the standard algorithm for converting a number N to its equivalent in base b, one performs the series of divisions

$$\begin{aligned} N/b &= q_1 + r_1/b & r_1 < b \\ q_1/b &= q_2 + r_2/b & r_2 < b \\ q_2/b &= q_3 + r_3/b & r_3 < b \\ &\vdots \\ q_{n-1}/b &= 0 + r_n/b & r_n < b \end{aligned}$$

The number in base b is then $r_n \ldots r_3 r_2 r_1$. *Eg* the octal equivalent of 61 decimal is 75:

$$\begin{aligned} 61/8 &= 7 + 5/8 \\ 7/8 &= 0 + 7/8 \end{aligned}$$

The following decimal print routine converts a 36-bit positive integer in accumulator T to decimal and types it out. The contents of T and T+1 are destroyed. The routine is called by a PUSHJ P,DECPNT where P is the pushdown pointer.

```
DECPNT: IDIVI   T,12            ;12(8) = 10(10)
        PUSH    P,T+1           ;Save remainder
```

```
        SKIPE   T               ;All digits formed?
        PUSHJ   P,DECPNT        ;No, compute next one
DECPN1: POP     P,T             ;Yes, take out in opposite order
        ADDI    T,60            ;Convert to ASCII (60 is code for 0)
        JRST    TTYOUT          ;Type out
```

This routine repeats the division until it produces a zero quotient. Hence it suppresses leading zeros, but since it is executed at least once it outputs one "0" if the number is zero. The TTYOUT routine returns with a POPJ P, to DECPN1 until all digits are typed, then to the calling program.

Space can be saved in the pushdown stack by storing the computed digits in the left halves of the locations that contain the jump addresses. This is accomplished in the decimal print routine by changing

PUSH P,T+1 to HRLM T+1,(P)

and

POP P,T to HLRZ T,(P)

The routine can handle a 36-bit unsigned integer if the IDIVI T,12 is replaced by

MACRO interprets a number following ↑D as decimal.

```
        LSHC    T,-↑D35         ;Shift right 35 bits into T+1
        LSH     T+1,-1          ;Vacate the T+1 sign bit
        DIVI    T,12            ;Divide double length integer by 10
```

Table Searching

Many data processing situations involve searching for information in tables and lists of all kinds. Suppose we wish to find a particular item in a table beginning at location TAB and containing *N* items. Accumulator T contains the item. The right half of A is used to index through the table, while the left half keeps a control count to signal when a search is unsuccessful.

```
        MOVSI   A,-N            ;Put -N, 0 in A
        CAMN    T,TAB(A)        ;Skip if current item not the one
        JRST    FOUND           ;Item found
        AOBJN   A,.-2           ;Try next item until left count = 0
        . . .                   ;Item not in list
```

The location of the item (if found) is indicated by the number in the right half of A (its address is that quantity plus TAB). A slightly different procedure would be

```
        HRLZI   A,-N
        CAME    T,TAB(A)        ;Skip if current item is the one
        AOBJN   A,.-1
        JUMPL   A,FOUND         ;Jump if left count < 0
        . . .                   ;Item not found
```

Locations used for a list can be scattered throughout memory if data is kept in the left half of each location and the right half addresses the next location in the list. The final location is indicated by a zero right half. The following routine finds the last half word item in the list. It is entered at FIND with the first location in the list addressed by the right half of accumulator T. At the end the final item is in T right.

```
        MOVE    T,(T)           ;Move next item to T
FIND:   TRNE    T,777777        ;Skip if AC right = 0
        JRST    .-2
        HLRZS   T               ;Move final item to right
```

The following counts the length of the list in accumulator CNT.

```
        MOVEI   CNT,0           ;Clear CNT
        JUMPE   T,OUT           ;Jump out if T contains 0
        HRRZ    T,(T)           ;Get next address
        AOJA    CNT,.-2         ;Count and go back
```

Double Precision Floating Point

The following are straightforward routines for handling double precision floating point arithmetic in software format, *ie* using single precision instructions, as would be required with a KA10 processor. [§2.6 *describes the floating point instructions.*]

These routines are given to show the mechanics of double precision floating point operations. They produce correct results in all ordinary circumstances, but do not handle pathological cases.

```
DFAD:   UFA     A+1,M+1         ;Sum of low parts to A+2
        FADL    A,M             ;Sum of high parts to A, A+1
        UFA     A+1,A+2         ;Add low part of high sum to A+2
        FADL    A,A+2           ;Add low sum to high sum
        POPJ    P,

DFSB:   DFN     A,A+1           ;Negate double length operand
        PUSHJ   P,DFAD          ;Call double floating add
        DFN     A,A+1           ;-(M - AC) = AC - M
        POPJ    P,

DFMP:   MOVEM   A,A+2           ;Copy high AC operand in A+2
        FMPR    A+2,M+1         ;One cross product to A+2
        FMPR    A+1,M           ;Other to A+1
        UFA     A+1,A+2         ;Add cross products into A+2
        FMPL    A,M             ;High product to A, A+1
        UFA     A+1,A+2         ;Add low part to cross sum in A+2
        FADL    A,A+2           ;Add low sum to high part of product
        POPJ    P,
```

A double precision division is of the form

$$\frac{A}{B} = \frac{a + c \times 2^{-27}}{b + d \times 2^{-27}}$$

Using the relationship

$$A/b = q + r \times 2^{-27}/b$$

where q and r are the quotient and remainder produced by FDVL, the following routine computes a double length quotient by the approximation

$$\frac{A}{B} \cong q + \frac{(r - qd) \times 2^{-27}}{b}$$

which gives a result correct to the next-to-last bit in the low order half.

```
DFDV:   FDVL    A,M         ;Get high part of quotient
        MOVN    A+2,A       ;Copy negative of quotient in A+2
        FMPR    A+2,M+1     ;Multiply by low part of divisor
        UFA     A+1,A+2     ;Add remainder
        FDVR    A+2,M       ;Divide sum by high part of divisor
        FADL    A,A+2       ;Add result to original quotient
        POPJ    P,
```

Proof: Using the expansion

$$\frac{1}{x+y} = \frac{1}{x}\left[1 - \frac{y}{x} + \frac{y^2}{x^2} - \frac{y^3}{x^3} + \ldots\right] \qquad (y^2 < x^2)$$

and letting $x = b$ and $y = d2^{-27}$ gives

$$\frac{A}{B} = \left(q + \frac{r2^{-27}}{b}\right)\left[1 - \frac{d2^{-27}}{b} + \frac{d^2 2^{-54}}{b^2} - \frac{d^3 2^{-81}}{b^3} + \ldots\right]$$

Multiplying out and gathering like terms gives

$$\frac{A}{B} = q + \frac{1}{b}(r - qd)2^{-27} - \frac{d}{b^2}(r - qd)2^{-54} + \frac{d^2}{b^3}(r - qd)2^{-81} - \ldots$$

where the first two terms on the right are those in the approximation given above.

The ratio of adjacent terms is

$$\frac{T_{n+1}}{T_n} = \frac{-d2^{-27}}{b}$$

In an alternating convergent series, the error due to truncation is smaller than the first term dropped. Therefore

$$|Error| < \frac{d2^{-27}}{b} T_n$$

Since the maximum value of d is less than 1 and the minimum value of b (normalized) is ½,

$$|Error| < T_n 2^{-26}$$

2.12 INPUT-OUTPUT

The input-output instructions govern all transfers of data to and from the peripheral equipment, and also perform many operations within the processor. An instruction in the in-out class is designated by 111 in bits 0–2, *ie* its left octal digit is 7. Bits 3–9 address the device that is to respond to the instruction. The format thus allows for 128 codes, two of which, 000 and 004 respectively, address the processor and priority interrupt, and are used for the console as well. The KA10 also uses the first two codes for the time share hardware, but the KI10 has a separate code, 010, for this purpose. A chart in Appendix A lists all devices for which codes have been assigned, and gives their mnemonics and DEC option numbers. Electrical and logical specifications of the IO bus are given in the interface manual.

Bits 13–35 are the same as in all other instructions: they are the *I*, *X*, and *Y* parts, which are used to calculate an effective address, set of conditions, or mask to be used in the execution of the instruction. The remaining bits, 10–12, select one of the following eight IO instructions.

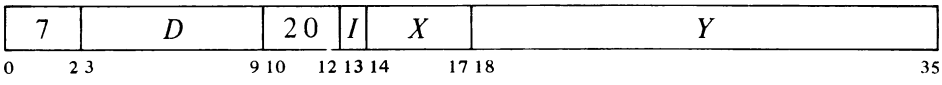
NOTE

All instructions described in the remainder of this manual are in-out instructions, which are affected by the time share restrictions. In the KI10 and KL10 no in-out instruction can be performed by a user mode program unless the User In-out flag is set. In the KI10, in-out instructions using device codes 740 and above are not restricted. But an instruction using a device code under 740 cannot be performed by a user mode program unless User In-out is set and cannot be performed in supervisor mode at all (in-out is normally handled in kernal mode). Any in-out instruction that violates these restrictions does not perform the functions given for it in the instruction description. Instead it acts just like an MUUO [§2.10].

These restrictions will not be mentioned in the instruction descriptions, as they apply to *all* instructions from this point on.

Input and output for system users is normally handled by the Monitor using MUUOs and various software formats. For information on user procedures vis-a-vis Monitor handling of user IO requirements, refer to Chapters 4–6 of *DECsystem-10 Monitor Calls*, manual DEC-10-MRR*x*-D.

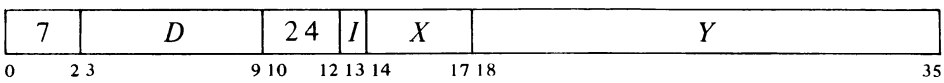
CONO Conditions Out



Set up device *D* with the effective initial conditions *E*. The number of condition bits in *E* that are actually used depends on the device.

E will always be regarded as being bits 18–35, even though it is actually placed on both halves of the bus and many devices receive the information from the left half.

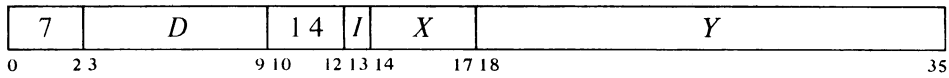
CONI Conditions In



Read the input conditions from device *D* and store them in location *E*. The

number of condition bits stored depends on the device; the remaining bits in location *E* are cleared.

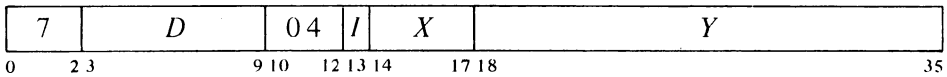
DATAO Data Out



Send the contents of location *E* to the data buffer in device *D*, and perform whatever control operations are appropriate to the device.

The amount of data actually accepted by the device depends on the size of its buffer, its mode of operation, etc. The original contents of location *E* are unaffected.

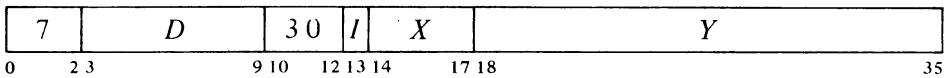
DATAI Data In



Move the contents of the data buffer in device *D* to location *E*, and perform whatever control operations are appropriate to the device.

The number of data bits stored depends on the size of the device buffer, its mode of operation, etc. Bits in location *E* that do not receive data are cleared.

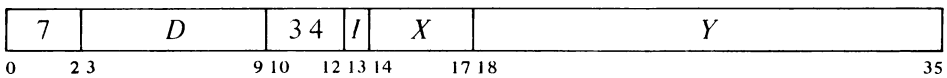
CONSZ Conditions In and Skip if Zero



Test the input conditions from device *D* against the effective mask *E*. If all condition bits selected by 1s in *E* are 0s, skip the next instruction in sequence.

If the device supplies more than 18 condition bits, only the right 18 are tested.

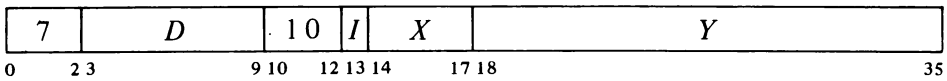
CONSO Conditions In and Skip if One



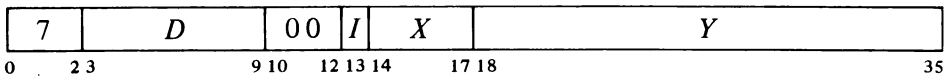
Test the input conditions from device *D* against the effective mask *E*. If any condition bit selected by a 1 in *E* is 1, skip the next instruction in sequence.

If the device supplies more than 18 condition bits, only the right 18 are tested.

BLKO Block Out



BLKI Block In



Add one to each half of a pointer in location E, and place the result back in E. Then perform a data IO instruction in the same direction as the block IO instruction, using the right half of the incremented pointer as the effective address. If the given instruction is a BLKO, perform a DATAO; if a BLKI, perform a DATAI.

A block IO instruction is effectively a whole in-out data handling subroutine. It keeps track of the block location, transfers each data word, and determines when the block is finished.

Initially the left half of the pointer contains the negative of the number of words in the block, the right half contains an address one less than that of the first word in the block.

The remaining actions taken by this instruction depend on whether it is executed as a priority interrupt instruction [§2.13].

♦ *Not as an Interrupt Instruction.* If the addition has caused the count in the left half of the pointer to reach zero, go on to the next instruction in sequence. Otherwise skip the next instruction.

♦ *As an Interrupt Instruction.* If the addition has caused the count in the left half of the pointer to reach zero, execute the instruction in the second interrupt location for the channel. Otherwise dismiss the interrupt and return to the interrupted program.

Note: The KA10 increments the two halves of the pointer by adding $1\,000001_8$ to the entire register. In the KI10 the two halves are handled independently.

The above eight instructions differ from one another in their total effect, but they are not all different with respect to any given device. A BLKO acts on a device in exactly the same way as a DATAO – the two differ only in counting and other operations carried out within the processor and memory. Similarly, no device can distinguish between a BLKI and a DATAI; and a device always supplies the same input conditions during a CONI, CONSZ or CONSO whether the program tests them or simply stores them.

Hence the eight instructions may be categorized as of four types, represented by the first four instructions described above. Moreover, a complete treatment of the programming of any device can be given in terms of these four instructions, two of which are for input and two for output. The four exhaust the types of information transfer that occur in the IO system, at least three of which are applicable to any given device. Thus all instruction descriptions in the rest of this manual will be of the CONO, CONI, DATAO and DATAI instructions combined with the various device codes. The discussion of each device will present timing information pertinent to device operation, as internal device timing is dependent only upon the device and not upon processor instruction time (which is given in Appendix D).

The word "input" used without qualification always refers to the transfer of data from the peripheral equipment into the processor; "output" refers to the transfer in the opposite direction.

Every device requires initial conditions; these are sent by a CONO, which

can supply up to eighteen bits of control information to the device control register. The program can determine the status of the device from up to thirty-six bits of input conditions that can be read by a CONI (but only the right eighteen can be tested by a CONSZ or CONSO). Some input bits simply reflect initial conditions sent by a previous CONO; others are set up by output conditions but are subject to subsequent adjustment by the device; and still others, such as status levels from a tape transport, have no direct connection with output conditions.

Data is moved in and out in characters of various sizes or in full 36-bit words. Each transfer between memory and a device data buffer requires a single DATAI or DATAO. Every device has a CONO and CONI, but it may have only one data instruction unless it is capable of both input and output. *Eg*, the paper tape reader has only a DATAI, the tape punch has only a DATAO, but the console terminal has both. (A high speed device, such as a disk file, can be connected to a direct-access processor, which in turn is connected directly to memory by a separate memory bus and handles data automatically. This eliminates the need for the program to give a DATAO or DATAI for each transfer.)

A DATAI that addresses an output-only device simply clears location *E*. DATAI PI, (code 70044) produces only this effect as the priority interrupt has no data for input. On the other hand a DATAO that addresses an input-only device is a no-op.

When the device code is undefined or the addressed device is not in the system, a DATAO, CONO or CONSO is a no-op, a CONSZ is an absolute skip, a DATAI or CONI clears location *E*.

A Typical IO Device. Every device has a 7-bit device selection network, a priority interrupt assignment, and at least two flags, Busy and Done, or some equivalent. The selection network decodes bits 3–9 of the instruction so that only the addressed device responds to signals sent by the processor over the in-out bus. To use the device with the priority interrupt, the program must assign a channel to it. Then whenever an appropriate event occurs in the device, it requests an interrupt on the assigned channel.

The Busy and Done flags together denote the basic state of the device. When both are clear the device is idle. To place the device in operation, a CONO or DATAO sets Busy. If the device will be used for output, the program must give a DATAO that sends the first unit of data – a word or character depending on how the device handles information. When the device has processed a unit of data, it clears Busy and sets Done to indicate that it is ready to receive new data for output, or that it has data ready for input. In the former case the program would respond with a DATAO to send more data; in the latter, with a DATAI to bring in the data that is ready. If an interrupt channel has been assigned to the device, the setting of Done signals the program by requesting an interrupt; otherwise the program must keep testing Done to determine when the device is ready.

Busy and Done both set is a meaningless situation.

All devices function basically as described above even though the number of initial conditions varies considerably. Besides Busy and Done flags, the tape reader and punch have a Binary flag that determines the mode of operation of the device with respect to the data it processes – alphanumeric or binary. The terminal has no binary flag, but it has two Busy flags and two Done flags – one pair for input, another for output. A complicated device, such as magnetic tape, may require two device codes to handle the large number of conditions associated with it. Initial conditions for a tape system include a transport address and an actual command the tape control is to perform; input conditions include error flags and transport status levels.

Occasionally a device with a second code may use a DATAI or DATAO to transmit additional control or maintenance information.

Most IO devices involve motion of some sort, usually mechanical (in a display only the electron beam moves). With respect to mechanical motion

there are two types of devices, those that stay in motion and those that do not. Magnetic tape is an example of the former type. Here the device executes a command (such as read, write, space forward) and the done flag indicates when the entire operation is finished. A separate data flag signals each time the device is ready for the program to give a DATAI or DATAO, but the tape keeps moving until an entire record or file has been processed.

Paper tape, on the other hand, stops after each transfer, but the program need not give a new CONO every time. The reader logic is set up so that a DATAI not only reads the data, but also clears Done and sets Busy. Hence if the instruction is given within a critical time, the tape moves continuously and only two CONOs are required for a whole series of transfers: one to start the tape, and one to stop it after the final DATAI.

Other devices operate in one or the other of these two ways but differ in various respects. The tape punch and terminal output are like the reader. Terminal input is initiated by the operator striking a key rather than by the program. The card reader reads an entire card on a single CONO, with a DATAI required for each column. The DECtape stays in motion, and the program must give a CONO to stop it or it will go all the way to the end zone.

Readin Mode

This mode of processor operation provides a means of placing information in memory without relying on a program already in memory or loading one word at a time manually. Its principal use is to read in a short loader program which is then used for loading other information. A loader program should ordinarily be used rather than readin mode, as a loader can check the validity of the information read.

Pressing the readin key on the console activates readin mode by starting the processor in a special hardware sequence that simulates a DATAI followed by a series of BLKI instructions, all of which address the device whose code is selected by the readin device switches at the left just above the console operator panel. Various devices can be used, and for each there are special rules that must be followed. But the readin mode characteristics of any particular device are treated in the discussion of the device. Here we are concerned only with the general characteristics.

At present readin is limited to paper tape, DECtape, and standard magnetic tape.

The information read is a block of data (such as a loader program) preceded by a pointer for the BLKI instructions. The left half of the pointer contains the negative of the number of words in the block, the right half contains an address one less than that of the location that is to receive the first word.

To read in, the operator must set up the device he is using, set its code into the readin device switches, and press the readin key. This key function first duplicates the action of the console reset key, which clears both the processor and the in-out equipment; in particular it places the processor in executive mode, and in the KI10 selects kernel mode with executive paging disabled, so all access will be to the first 256K of physical memory unpaged. Following this the processor places the device in operation, brings the first

word (the pointer) into location 0, and then reads the data block, placing the words in the locations specified by the pointer. Data can be placed anywhere in the first 256K of memory (including fast memory) except in location 0. The operation affects none of memory except location 0 and the block area.

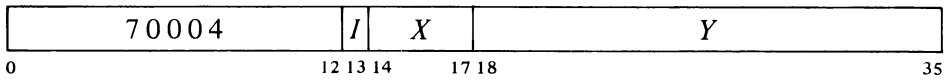
Upon completing the block, the processor leaves readin mode and begins normal operation. This is done in the KI10 by jumping to the location containing the last word in the block, in the KA10 by executing the last word as an instruction. In the KA10 the processor stops after executing the first instruction if the single instruction switch is on.

Console-Program Communication

Neither the processor nor the priority interrupt system require all four types of IO instructions, so the program can make use of their device codes for communicating with the console. Both processors have two instructions that transfer data between console and program. But in the KI10, the program can actually operate some of the switches on the console. For this purpose it uses a data-out instruction with the device code for the paper tape reader (an input-only device). The KI10 program can also inspect the states of a number of operating and sense switches, but the bits for these are included in the left half words of the standard input conditions for the interrupt and processor [§§2.13, 2.14].

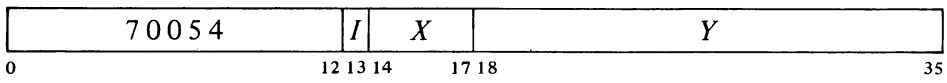
MACRO also recognizes the mnemonic RSW (Read Switches) as equivalent to DATAI APR,.

DATAI APR, Data In, Console



Read the contents of the console data switches into location *E*.

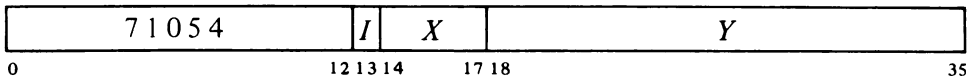
DATAO PI, Data Out, Console



Unless the console MI program disable switch is on, display the contents of location *E* in the console memory indicators and turn on the triangular light beside the words PROGRAM DATA just above the indicators (turn off the light beside MEMORY DATA).

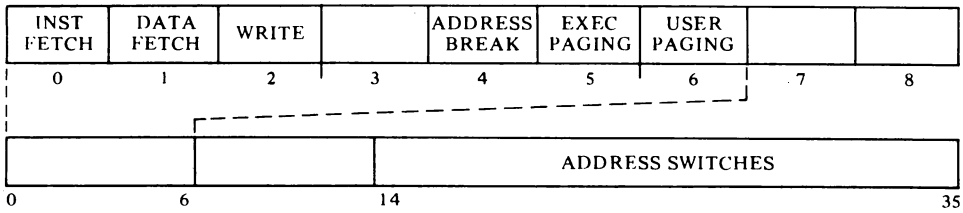
Once the indicators have been loaded by the program, no address condition selected from the console [*Appendix F*] can load them until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key.

DATAO PTR, Operating Data Out, Console



Unless the MI program disable switch is on, set up the console address and address-condition switches according to the contents of location *E* as shown (a 1 in a bit turns on the switch, a 0 turns it off).

On the KI10 console, all switches are pushbutton-flipflop combinations; the instruction of course controls the flipflops, not the buttons.



For complete information on the use of these switches, see Appendix F1.

2.13 PRIORITY INTERRUPT

Most in-out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them. These devices however must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often result in loss of information and certainly results in operating the device below its maximum speed. The priority interrupt is designed with these considerations in mind, *ie* the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The hardware also allows conditions internal to the processor to signal the program by requesting an interrupt. Programmable interrupt requests are handled through seven channels arranged in a priority chain, with assignment of devices to channels entirely at the discretion of the programmer. To assign a device to a channel, the program sends the number of the channel to the Device Control Register as a part of the conditions given by a CONO (usually bits 33-35). Programmable channels are numbered 1-7, with 1 having the highest priority. A zero assignment disconnects the device from the interrupt channels altogether. Any number of devices can be connected to a single channel and some can be connected to two channels (*eg* a device may signal that data is ready on one channel, that an error has occurred on another). The 10-11 interface (DTE20), which connects the KL10 Processor to the PDP-11 Processor, can request an interrupt at priority level 0. This is a fixed channel assignment which is always on in the DTE20. Thus the DTE20 can interrupt the KL10 at any time and whether the PI System is turned on or not. The level zero interrupt is used solely during the implementation of examine (or PI DATAO) and deposit (or PI DATAI) functions via the DTE20.

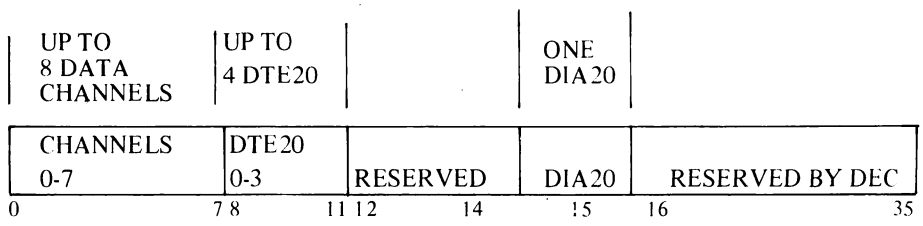
The request signal is generally derived from a flag that is set by various conditions in the device. Often associated with these flags are enabling flags, where the setting of some device condition flag can request an interrupt on the assigned channel only if the associated enabling flag is also set. The enabling flags are in turn controlled by the conditions supplied to the device by a CONO. *Eg* a device may have half a dozen flags to indicate various internal conditions that may require service by an interrupt; by setting up the associated enabling flags, the program can determine which conditions shall actually request interrupts in any given circumstances.

When a device requires service it sends an interrupt request signal over the system in-out bus to its assigned channel in the processor. The processor accepts the request depending upon certain conditions; such as, that the channel must be active (on for programmable channels). The request signal is a level so it remains on the bus until turned off by the program (CONO, DATAO or DATAI), depending upon the device. Thus, if a request is not accepted when made, it will be accepted when the conditions are satisfied. A single channel will shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request.

The program can usually trigger a request from a device but delay its acceptance by turning on the channel later. Having accepted a request, the processor will do nothing further with it unless the Priority Interrupt System is on. But even if the system is off, the processor will continue to accept requests on the other channels, and when the system is finally turned on it will respond as though all requests had just been accepted, handling the highest priority one first. The way in which the interrupts are handled, the conditions that affect them, and so forth depend upon the type of processor.

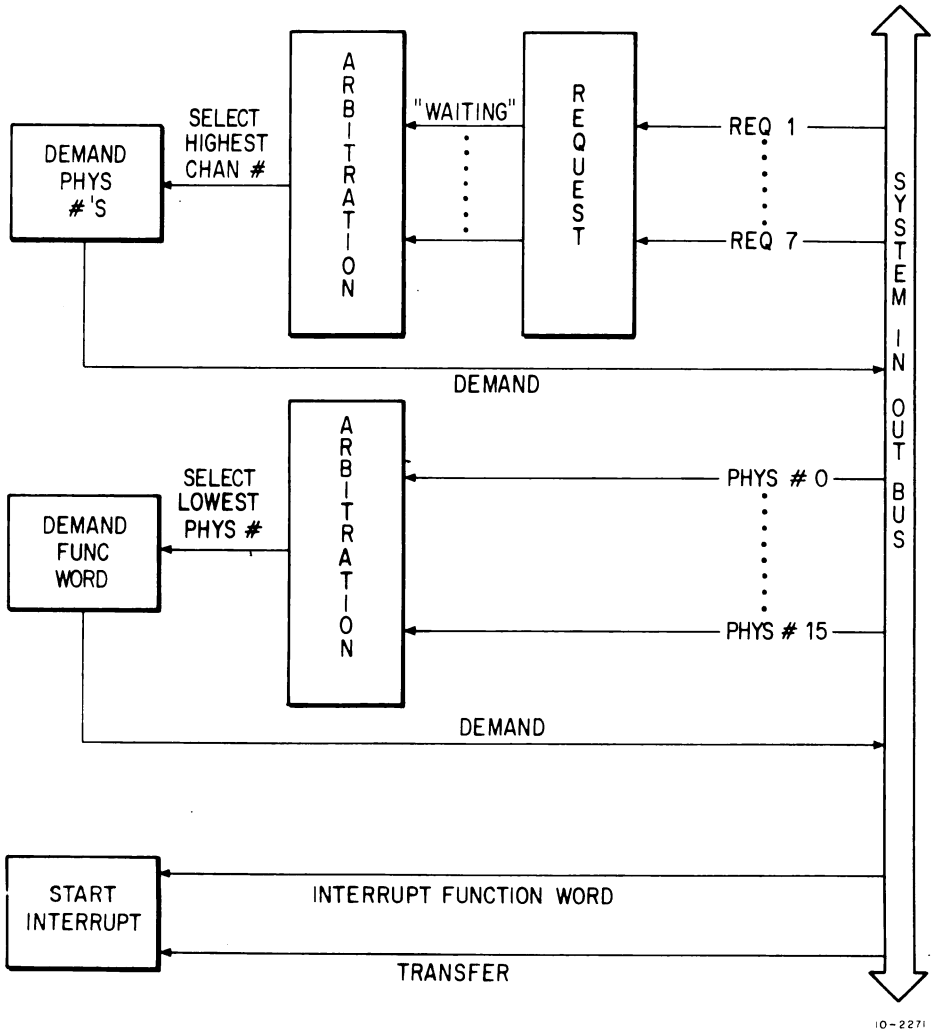
KL10 Interrupt

A request made to an active channel is accepted immediately unless some channel is already waiting for an interrupt to start or an interrupt is starting for some channel. When an interrupt request is detected by the processor, with the system on, the channel must wait for the interrupt to start. The processor will however delay any action on the request if it is already holding an interrupt for the same channel or for a channel with higher priority than those on which requests have been accepted (in other words, if the current program is a higher priority interrupt routine). When a waiting channel has priority higher than the current program, the processor sends an interrupt demand signal for the waiting channel(s) that has highest priority. This action makes use of the system in-out bus. Should the bus be busy, the demand is sent as soon as the bus becomes available, taking precedence over any IO instruction that may be waiting too (note that in this situation, the program actually stops). The demand signal goes out on the bus in parallel to all devices on the selected channel. Upon receiving the demand signal, those devices interrupting on the selected channel place their hardwired physical device numbers onto the system in-out bus in the prescribed bit positions. Bit assignments for the physical numbers are indicated below.

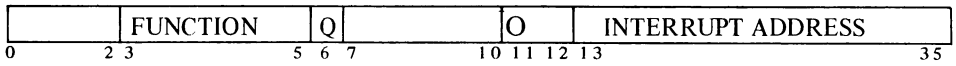


Upon receiving the physical numbers from those devices on the channel of highest priority, the processor selects the discrete device among these with the numerically lowest physical number to be the highest overall priority. Remember that there are two orders of priority associated with the interrupt.

First there is the channel, and secondly for all devices requesting interrupts simultaneously on the same channel physical number priority. Once again the processor sends the interrupt demand signal for the waiting channel that has highest overall priority, *eg* highest channel and lowest physical number. Upon receipt of the interrupt demand signal, the selected device now sends an interrupt function word back to the processor together with a transfer signal. Upon receipt of the function word and transfer, the processor stops the current program at the first allowable point to begin an interrupt for the waiting channel. Allowable stopping points are at the completion of an instruction, following the retrieval of an address word in an effective address calculation of any kind (including an indirect word reference specified in the pointer word of a byte instruction), between transfers in a BLT, and while an IO instruction is waiting for the bus.



When an interrupt starts, PC points to the interrupted instruction so that a proper return can later be made from the interrupted program. The action taken by the processor in starting an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the Executive Process Table are associated with each channel: locations 40+2*N* and 41+2*N*, where *N* is the channel number, channel 1 uses locations 42 and 43, channel 2 uses locations 44 and 45, and so on to channel 7 which uses locations 56 and 57. The processor starts a standard interrupt for channel *N* by executing the instruction in the first interrupt location for the channel, *ie* location 40+2*N*. The fixed locations, however, need not be used. The interrupt function word sent by the device may specify a vector (or DISPATCH) interrupt using a pair of locations specified by the function word, or some other interrupt function entirely. The format of the function word and the operations the processor performs in response to the function selected in bits 3-5 of the word are as follows:



Bits 0-2 Interrupt Address Space

0 Executive Process Table

1 Executive Virtual Address Space

2 Not Used – Reserved by DEC

3 Not Used – Reserved by DEC

4 Physical Address Space

5 Not Used – Reserved by DEC

6 Not Used – Reserved by DEC

7 Not Used – Reserved by DEC

Bits 3-5 Interrupt Function

0 No response from device. Execute the instruction in location 40+2*N* of Executive Process Table.

1 Standard Interrupt. Execute the instruction in location 40+2*N* of Executive Process Table.

2 Dispatch. Execute the instruction in the location specified by bits 14-35.

3 Increment. If Q is zero, increment the contents of the location specified by bits 14-35.

If Q is one, decrement the contents of the location specified in bits 14-35.

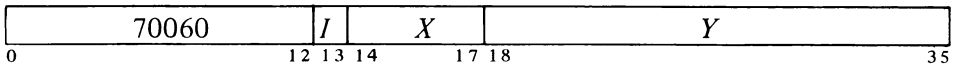
4 DATAO. Perform a DATAO using the contents of bits 14-35 as the effective address.

5 DATAI. Perform a DATAI using the contents of bits 14-35 as the effective address.

6 Reserved by DEC. Used only by DTE20 Byte Transfers.

7 Reserved by DEC. Not used at this time.

CONO PI, Conditions Out Priority Interrupt



Perform the functions specified by the effective conditions *E* as shown (a 1 in a bit produces that indicated function, a 0 has no effect).

Bit	*Function*
18	Write Even – Address Parity
19	Write Even – Data Parity
20	Write Even – Directory Parity
21	
22	Drop program requests on selected channels.
23	Clear PI System.
24	Initiate interrupts on the selected channels.
25	Turn on the selected channels.
26	Turn off the selected channels.
27	Deactivate the PI System.
28	Activate the PI System.
29-35	Select channels for bits 22, 24, 25, and 26.

Bits 18-20 are intended for test purposes only to allow the processor to provoke errors.

Bit 28+*N* selects channel *N*.

22 On channels selected by 1s in bits 29-35, turn off any interrupt requests made previously by the program (via bit 24).

23 Deactivate the priority interrupt system, turn off all channels, eliminate all interrupt requests that have already been accepted but are still waiting, and dismiss all interrupts that are currently being held.

24 Request interrupts on channels selected by 1s in bits 29-35, and force the processor to accept them even on channels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given channel another is started, until the request is turned off by a CONO that selects the same channel and has a 1 in bit 22.

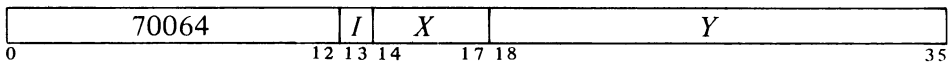
25 Turn on the channels selected by 1s in bits 29-35 so interrupt requests can be accepted on them.

26 Turn off the channels selected by 1s in bits 29-35 so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

27 Deactivate the priority interrupt system. The processor can then accept requests, but it can neither start nor dismiss an interrupt.

28 Activate the priority interrupt system so the processor can accept requests and can start, hold, and dismiss interrupts.

CONI PI, CONDITIONS IN PRIORITY INTERRUPT



Read the status of the priority interrupt (including write even parity bits) into location *E* as shown.

Bit	*Function*
1-10	
11	Program request on channel 1.
12	Program request on channel 2.
13	Program request on channel 3.
14	Program request on channel 4.
15	Program request on channel 5.
16	Program request on channel 6.
17	Program request on channel 7.
18	Write even Address Parity.
19	Write even Data Parity.
20	Write even Directory Parity.
21-27	Interrupts holding on channels 1-7.
28	PI System on.
29-35	Active channels 1-7.

Channels that are active are indicated by 1s in bits 29-35: 1s in bits 21-27 indicate channels on which interrupts are currently being held. 1s in bits

11-17 indicate channels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on. The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18-20 reflect the state of the three write even parity hardware flags which may be enabled or disabled by a CONO PI instruction.

Processor Conditions. There are a number of conditions internal to the processor that can signal the program by requesting an interrupt on the channel assigned to the processor. Condition IO instructions are used to control the appropriate flags and to inspect other internal conditions of interest ot the program.

KI10 Interrupt

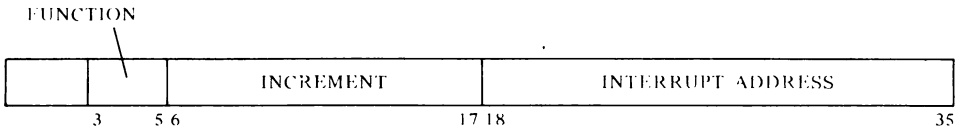
A request made to an active channel is accepted immediately unless some channel is already waiting for an interrupt to start or an interrupt is starting for some channel. Once a request is accepted with the system on, the channel must wait for the interrupt to start. The processor however will delay any action on the request if it is already holding an interrupt for the same channel or for a channel with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). When a waiting channel has priority higher than the current program, the processor sends an interrupt-granted signal for the waiting channel that has highest priority. This action makes use of the IO bus. Should the bus be busy, the grant is sent as soon as the bus becomes available, taking precedence over any IO instruction that may also be waiting (note that in this situation the program actually stops). The grant signal goes out on the bus and is transmitted serially from one device to the next. Upon receiving the grant, a device that is not requesting an interrupt on the specified channel sends the signal on to the next device. A device that is requesting an interrupt on the specified channel terminates the signal path and sends an interrupt function word back to the processor.

Note that there are therefore two orders of priority associated with an interrupt: first the channel, and then for all devices requesting interrupts simultaneously on the same channel, proximity to the processor on the bus. For priority purposes, all devices on the left bus are closer than those on the right bus.

Upon receipt of the function word, the processor stops the current program at the first allowable point to start an interrupt for the waiting channel for which the grant was made. Allowable stopping points are at the completion of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), between transfers in a BLT, between steps in the calculation of the first part of the quotient in double floating division, and while an IO instruction is waiting for the bus. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

The action taken by the processor in starting an interrupt depends upon the function specified by the function word returned to the processor. Two fixed locations in the executive process table are associated with each channel: locations $40+2N$ and $41+2N$, where N is the channel number. Channel 1 uses locations 42 and 43, channel 2 uses 44 and 45, and so on to

channel 7 which uses 56 and 57. The processor starts a "standard" interrupt for channel *N* by executing the instruction in the first interrupt location for the channel, *ie* location 40 + 2*N*. The fixed locations however need not be used. The interrupt function word sent by the device may specify a standard interrupt using the fixed locations, or an equivalent interrupt using a pair of locations specified by the function word, or some other interrupt function entirely. The format of the function word and the operations the processor performs in response to the function selected by bits 3–5 of the word are as follows.



Bits 3–5	*Interrupt Function*
0	Processor waiting. If no response, perform a standard interrupt (see function 1).
1	Standard interrupt – execute the instruction in location 40 + 2*N* of the executive process table.
2	Dispatch – execute the instruction in the location specified by bits 18–35.
3	Increment – add the contents of bits 6–17 to the contents of the location specified by bits 18–35. The increment is a fixed point number in twos complement notation, bit 6 being the sign, and bit 17 corresponding to bit 35 of the memory word.
4	DATAO – do a DATAO for this device using the contents of bits 18–35 as the effective address.
5	DATAI – do a DATAI for this device using the contents of bits 18–35 as the effective address.
6	Not used – reserved by DEC.
7	Not used – reserved by DEC.

A device designed originally for use with the KA10 will work when connected to the KI10 bus, where it always requests a standard interrupt by providing no response to the grant. Note that for simultaneous requests on a given channel, all KI10 devices that return a function word have priority over all KA10 devices and over any KI10 devices that do not return a function word. The last group includes the reader, punch and teletypewriter, which are contained in the processor, as well as the processor itself acting as a device [*see processor conditions*, §2.14].

At present, functions 6 and 7 produce standard interrupts.

Regardless of what mode the processor is in when an interrupt occurs, the interrupt operations are performed in kernel mode. No interrupt operation can set Overflow or either of the trap flags; hence an overflow trap can never occur as a direct result of an interrupt. A page failure that occurs in an interrupt operation is never trapped; instead it sets the In-out Page Failure flag, which requests an interrupt on the channel assigned to the processor [§2.14]. These considerations of course do not apply to a service routine called by an interrupt instruction.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being per-

formed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in the first or second interrupt location for a channel, in direct response by the hardware (rather than by the program) to a request on that channel. §2.12 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. The interrupt instructions executed in a standard or dispatch interrupt fall into three categories.

These locations may be the fixed ones for a standard interrupt or those given by the function word for a dispatch interrupt.

♦ *AOSX, SKIPX, SOSX, CONSX, BLKX.* If the skip condition specified by the instruction is satisfied, the processor dismisses the interrupt and returns immediately to the interrupted program (*ie* it returns control to the unchanged PC). If the skip condition is not satisfied, the processor executes the instruction contained in the second interrupt location.

Satisfaction of the condition does not change PC, as this would skip the next instruction in the interrupted program. In effect the instruction skips back to the interrupted program by skipping the second interrupt location.

Note that the interpretation of a BLKI or BLKO as a skip instruction is consistent with the description given in §2.12, the condition being that the count is not zero.

CAUTION

In the second interrupt location, a skip instruction whose condition is not satisfied hangs up the processor, which will keep repeating the instruction until the condition is satisfied.

♦ *JSR, JSP, PUSHJ, MUUO.* The processor holds an interrupt on the channel, takes the next instruction from the location specified by the jump (as indicated by the newly changed PC), and enters either kernel mode or the mode specified by the new PC word of the MUUO. Hence the instruction is usually a jump to a service routine handled by the Monitor.

♦ *All Other Instructions.* In general the processor simply executes the instruction, dismisses the interrupt, and then returns to the interrupted program. If the instruction is a jump (other than those mentioned above). the processor jumps to the newly specified location; but it dismisses the interrupt and returns to the mode it was already in when the interrupt occurred. Hence it effectively returns to the interrupted program but in a different place, and the original contents of PC are lost.

Since the interrupt operations are performed in kernel mode regardless of the actual mode of the processor, an XCT is performed as an executive XCT [§2.15]. The ultimate effect of the XCT depends of course on the instruction executed – and its effect is as described here for the various categories.

CAUTION

Neither an LUUO, a BLT, a DMOVEM, nor a DMOVNM will function in a reasonable manner as an interrupt instruction. Therefore do not use them.

Dismissing an Interrupt. Unless the interrupt operation dismisses the interrupt automatically, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority channel. Thus interrupts can be held on a number of channels simultaneously, but from the time an interrupt is started until it is dismissed,

no interrupt can be started on that channel or any channel of lower priority (requests, however, can be accepted on lower priority channels).

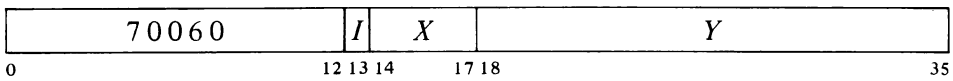
A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be on when the JEN is given). This instruction restores the channel on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority channels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, PUSHJ, or MUUO. If flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

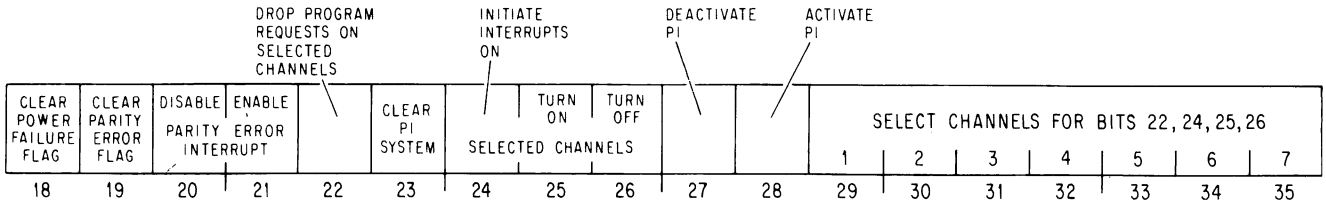
An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its channel and all channels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Priority Interrupt Conditions. The program can control the priority interrupt system by means of condition IO instructions. The device code is 004, mnemonic PI.

CONO PI, Conditions Out, Priority Interrupt



Perform the functions specified by the effective conditions *E* as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



20 Prevent the setting of the Parity Error flag from requesting an interrupt on the channel assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the channel assigned to the processor.

22 On channels selected by 1s in bits 29–35, turn off any interrupt requests made previously by the program (via bit 24).

23 Deactivate the priority interrupt system, turn off all channels, eliminate all interrupt requests that have already been accepted but are still waiting, and dismiss all interrupts that are currently being held.

Bits 18–21 are actually for processor conditions [§2.14].

24 Request interrupts on channels selected by 1s in bits 29–35, and force the processor to accept them even on channels that are off. The request remains indefinitely, so as soon as an interrupt is completed on a given channel another is started, until the request is turned off by a CONO that selects the same channel and has a 1 in bit 22.

Remember that the processor allows the program to continue while it grants an interrupt. Thus when this bit forces acceptance of a request, another program instruction or two may be performed before the interrupt, even on the highest priority channel. Moreover if the request is allowed to remain, additional instructions may be performed between successive interrupts.

For other than the highest priority channel, the greater the number of higher priority channels active, the greater the amount of program time available both initially and between successive interrupts. If the program forces an interrupt on the lowest priority channel when all are active, there can be as much as 40 μs of program time between the CONO PI, and its interrupt.

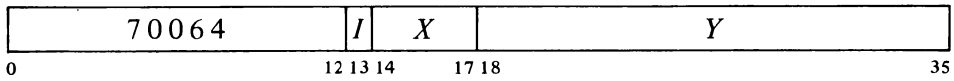
25 Turn on the channels selected by 1s in bits 29–35 so interrupt requests can be accepted on them.

26 Turn off the channels selected by 1s in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

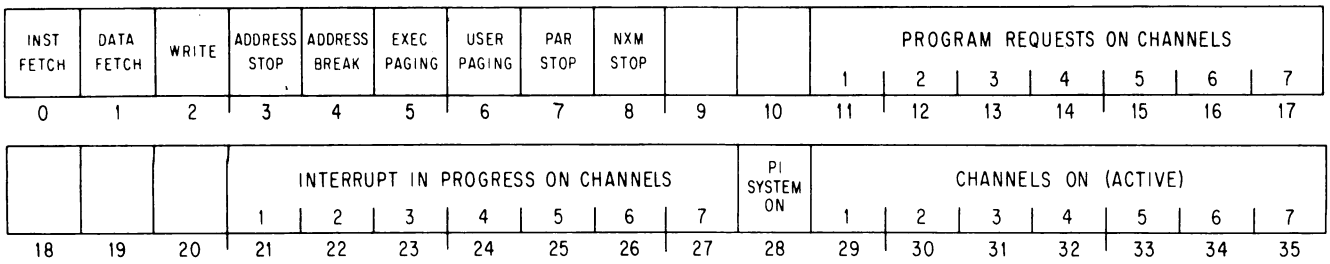
27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.

28 Activate the priority interrupt system so the processor can accept requests and can start, hold and dismiss interrupts.

CONI PI, Conditions In, Priority Interrupt



Read the status of the priority interrupt (and nine console operating switches) into location *E* as shown.



Channels that are active are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate channels on which interrupts are currently being held; 1s in bits 11–17 indicate channels that are receiving interrupt requests generated by a CONO PI, with a 1 in bit 24. A 1 in bit 28 means the interrupt system is on.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 0–8 reflect the settings of various console operating switches; for information on these switches refer to Appendix F1.

Timing. The time a device must wait for an interrupt to start depends on the number of channels in use, and how long the service routines are for devices on higher priority channels. If only one device is using interrupts, it need never wait longer than 10 μs.

Special Considerations. On a return to an interrupted program, the processor always starts the interrupted instruction over from the beginning. This causes special problems in a BLT and in byte manipulation.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two-part byte instruction. When this happens, First Part Done is set. This flag is saved as bit 4 of a PC word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an ILDB or IDPB would skip a byte. And if the routine restores the flag, the interrupted ILDB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In-out flag is set, a user program generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- No requests can be accepted, not even on higher priority channels, while an interrupt is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- Most in-out devices are designed to drop an interrupt request when the program responds, usually with a DATAI or DATAO. If an interrupt is handled neither by a BLKI or BLKO interrupt instruction nor by a service routine, the programmer must make sure the device is configured to drop the request on receipt of whatever response the program does give.
- The interrupt instruction that calls the routine must save PC if there is to be a return to the interrupted program. Generally a JSR is used as it saves both PC and the flags, and it uses no accumulator.
- The principal function of an interrupt routine is to respond to the situation that caused the interrupt. *Eg* computations that can be performed outside the routine should not be included within it.

♦ If the routine uses a UUO it must first save the contents of the pair of locations that will be changed by it in case the interrupted program was in the process of handling a UUO of the same type. For an MUUO the routine must save locations 424 and 425 of the user process table. For an LUUO the routine must save location 40 in the executive process table and the location used by the UUO handler instruction to store the PC word.
♦ The routine must dismiss the interrupt (with a JEN) when returning to the interrupted program. The flags and UUO locations should be restored.

KA10 Interrupt

A request made to an active channel is accepted at the next memory access unless the processor is starting an interrupt for any channel or holding an interrupt for the same channel. Once a request is accepted with the system on, the channel must wait for the interrupt to start. The processor however cannot start an interrupt if it is already holding an interrupt for a channel with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). When there is a higher priority channel waiting, the processor stops the current program at the first allowable point to start an interrupt for the waiting channel that has highest priority. Allowable stopping points are following the retrieval of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), and between transfers in a BLT. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

Interrupt locations for a second processor on the same memory are $140+2N$ and $141+2N$.

Two memory locations are associated with each channel: unrelocated locations $40+2N$ and $41+2N$, where N is the channel number. Channel 1 uses locations 42 and 43, channel 2 uses 44 and 45, and so on to channel 7 which uses 56 and 57. The processor starts an interrupt for channel N by executing the instruction in location $40+2N$. Even though the processor may be in user mode when an interrupt occurs, interrupt instructions are performed in executive mode.

Interrupt Instructions. An instruction executed in response to an interrupt request and not under control of PC is referred to elsewhere in this manual as being "executed as an interrupt instruction." Some instructions, when so executed, have different effects than they do when performed in other circumstances. And the difference is not due merely to being performed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed in location $40+2N$ or $41+2N$, in direct response by the hardware (rather than by the program) to a request on channel N. §2.12 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is not "executed as an interrupt instruction" even in the unlikely event the routine is stored within the interrupt locations and the BLKO is executed by an XCT. There are two categories of interrupt instructions.

♦ *Non-IO Instructions.* After executing a non-IO interrupt instruction, the processor holds an interrupt on the channel and returns control to PC.

Hence the instruction is usually a jump to a service routine. If the processor is in user mode and the interrupt instruction is a JSR, JSP, PUSHJ, JSA or JRST, the processor leaves user mode (the Monitor thus handles all interrupt routines [§2.16]).

If the interrupt instruction is not a jump, the processor continues the interrupted program while holding an interrupt – in other words it now treats the interrupted program as an interrupt routine. *Eg* the instruction might just move a word to a particular location. Such procedures are usually reserved for maintenance routines or very sophisticated programs.

♦ *Block or Data IO Instructions.* One or the other of two actions can result from executing one of these as an interrupt instruction.

If the instruction in $40 + 2N$ is a BLKI or BLKO and the block is not finished (*ie* the count does not cause the left half of the pointer to reach zero), the processor dismisses the interrupt and returns to the interrupted program. The same action results if the instruction is a DATAI or DATAO.

If the instruction in $40 + 2N$ is a BLKI or BLKO and the count does reach zero, the processor executes the instruction in location $41 + 2N$. This *cannot* be an IO instruction and the actions that result from its execution as an interrupt instruction are those given above for non-IO instructions.

CAUTION

The execution, as an interrupt instruction, of a CONO, CONI, CONSO or CONSZ in location $40 + 2N$ or *any* IO instruction in location $41 + 2N$ hangs up the processor.

Dismissing an Interrupt. Automatic dismissal of an interrupt occurs only in a DATAI or DATAO, or in a BLKI or BLKO with an incomplete block. Following any non-IO interrupt instruction, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority channel. Thus interrupts can be held on a number of channels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that channel or any channel of lower priority (requests, however, can be accepted on lower priority channels).

A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be on when the JEN is given). This instruction restores the channel on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority channels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, or PUSHJ. If flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

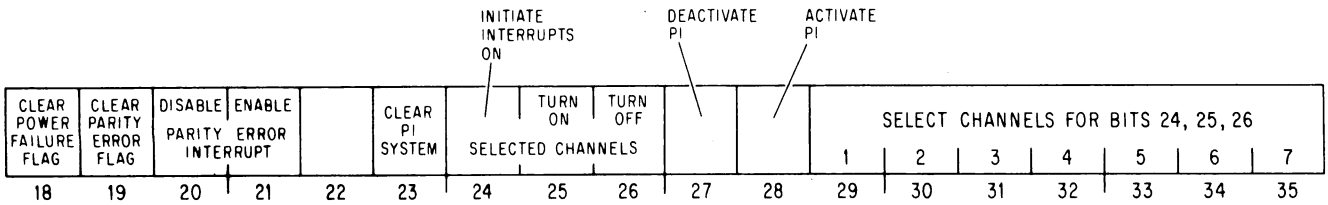
An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its channel and all channels of lower priority will be

disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Interrupt Conditions. The program can control the interrupt system by means of condition IO instructions. The device code is 004, mnemonic PI.

CONO PI, Conditions Out, Priority Interrupt

Perform the functions specified by the effective conditions *E* as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



Bits 18–21 are actually for processor conditions [§2.14].

20 Prevent the setting of the Parity Error flag from requesting an interrupt on the channel assigned to the processor.

21 Enable the setting of the Parity Error flag to request an interrupt on the channel assigned to the processor.

23 Deactivate the priority interrupt system, turn off all channels, eliminate all interrupt requests that have already been accepted but are still waiting, and dismiss all interrupts that are currently being held.

24 Request interrupts on channels selected by 1s in bits 29–35, and force the processor to accept them even on channels that are off. There is at most one interrupt on a given channel, and a request is lost if it is made by this means to a channel on which an interrupt is already being held.

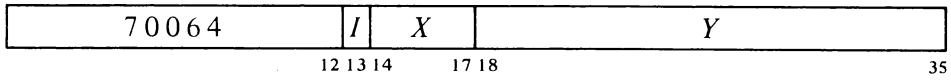
25 Turn on the channels selected by 1s in bits 29–35 so interrupt requests can be accepted on them.

26 Turn off the channels selected by 1s in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.

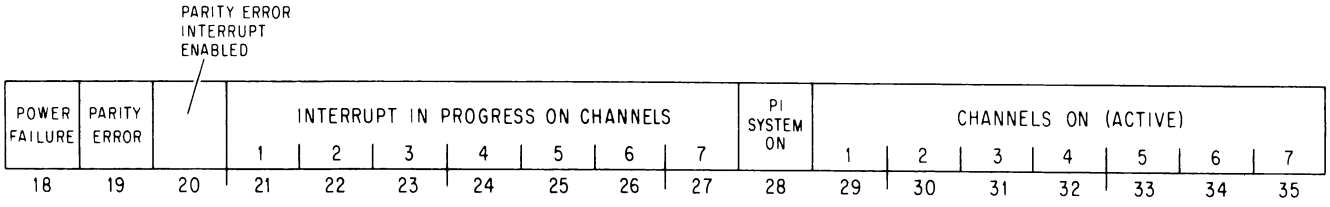
27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.

28 Activate the priority interrupt system so the processor can accept requests and can start, hold and dismiss interrupts.

CONI PI, Conditions In, Priority Interrupt



Read the status of the priority interrupt (and several bits of processor conditions) into location *E* as shown.



Channels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate channels on which interrupts are currently being held. A 1 in bit 28 means the interrupt system is on.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 18–20 actually read processor status condition [§2.14] as follows.

18 Ac power has failed. The program should save PC, the flags and fast memory in core, and halt the processor.

The setting of this flag requests an interrupt on the channel assigned to the processor. If the flag remains set for 5 ms, the processor is cleared.

Note that PC may point to an interrupt service routine rather than the main program.

19 A word with even parity has been read from core memory. If bit 20 is set, the setting of the Parity Error flag requests an interrupt on the channel assigned to the processor, at which time PC points to the instruction being performed or to the one following it.

Timing. The time a device must wait for an interrupt to start depends on the number of channels in use, and how long the service routines are for devices on higher priority channels. If only one device is using interrupts, it need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is made. The maximum time can be considered to be about 15 μs for FDVL, but a ridiculously long shift could take over 35 μs.

Special Considerations and Programming Suggestions. If the interrupt routine uses a UUO it must first save the contents of the pair of locations that will be changed by it in case the interrupted program was in the process of handling a UUO. Hence the routine must save unrelocated location 40 and the location used by the UUO handler instruction to store the PC word. In all other respects, the special considerations and programming suggestions given at the end of the section on the KI10 interrupt hold for the KA10.

2.14 PROCESSOR CONDITIONS

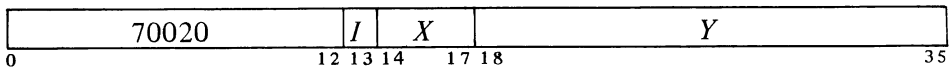
There are a number of internal conditions that can signal the program by requesting an interrupt on a channel assigned to the processor. Condition IO instructions are used to control the appropriate flags and to inspect other internal conditions of interest to the program.

KL10 Processor Conditions

In the KL10 all processor conditions act through the interrupt on a single channel assigned to the processor.

The processor device code is 000, Mnemonic APR.

CONO APR, Conditions Out Arithmetic Processor



Assign the interrupt channel specified by bits 33-35 of the effective conditions *E* and perform the functions specified by bits 18-32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

The following is a list of the bit assignments for the CONO APR:

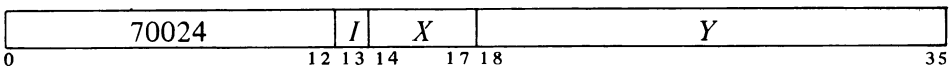
Bit	*Function*
18	
19	Clear all in-out devices.
20	Enable interrupt for flags selected in bits 24-31.
21	Disable interrupt for flags selected in bits 24-31.
22	Clear flags selected by bits 24-31.
23	Set flags selected by bits 24-31.
24	SBUS Error.
25	NXM Error.
26	In-Out page failure.
27s	MB Parity Error.
28	Cache Directory Parity Error.
29	Cache Address Parity Error.
30	Power Fail.
31	Sweep Done.

Setting these flags of course does not cause the conditions indicated, but may be used for testing the code which handles those conditions.

32	0
33-35	Priority interrupt assignment.

A1 in bit 19 produces the IO reset signal, which clears the control logic in all of the peripheral equipment (but neither the priority interrupt system nor the processor conditions).

CONI APR, Conditions-In Arithmetic Processor



Read the status of the processor into location *E* as shown.

Bit	*Function*
0-5	0
6	SBUS Error
7	Non-Existent Memory Error
8	In-Out Page Failure
9	Memory Buffer Parity Error
10	Cache Directory Error
11	Cache Address Parity Error
12	Power Fail
13	Sweep Done
14-18	0
19	Sweep Busy
20	
21	
22	
23	
24	SBUS Error
25	Non-Existent Memory Error
26	In-Out Page Failure
27	Memory Buffer Parity Error

Bits 6-13 indicate which flags are enabled to interrupt on the APR channel specified in bits 33-35 of the CONI APR word.

Bits 24-31 are the APR flags; they request interrupts only if enabled.

28 Cache Directory Error

29 Cache Address Parity Error

30 Power Fail

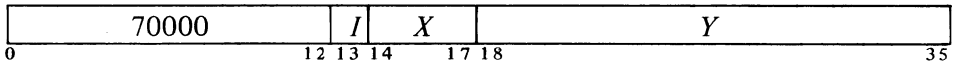
31 Sweep Done

32 Interrupt Request

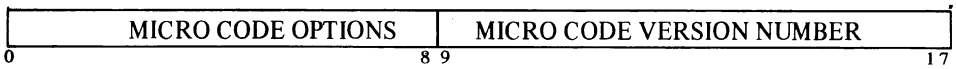
33-35 Priority Interrupt Assignment

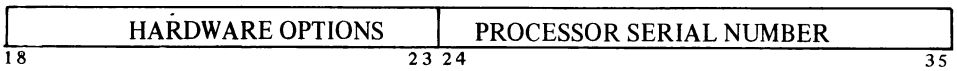
Interrupts are requested on the APR channel (assigned by bits 33-35 of the CONO) by the setting of the following enabled flags: Power Fail, In-Out Page Failure, Sweep Done, SBUS Error, Non-Existent Memory Error, MB Parity Error, Cache Directory Error, and Cache Address Parity Error. Bit 19 indicates that a sweep instruction is in progress, *ie* when the sweep instruction completes Sweep Busy clears and Sweep Done sets.

APRID, Identification Arithmetic Processor



Read the Micro Code Options and Micro Code Version Number into the left half of location *E* and the Hardware Options and Processor Serial Number into the right half of location *E* as shown.



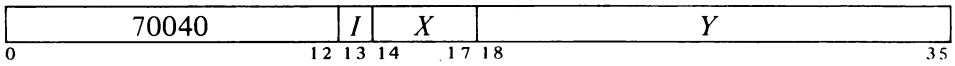


The processor has an internal register called the Error Address Register, or ERA. This register contains the last address requested from the Memory System along with Error Flags indicating what kind of operation (read, write, etc.) had been requested of the memory. The ERA Register holds on an error condition signaled by the memories and is never changed until this instruction is performed to read the appropriate error information contained in the ERA. Having done this, the software indicates that it has done so and the ERA Register is free to record the next error that may occur. In other words, until an error occurs, the ERA Register is changing with every Memory Address that is being sent by the processor to the Memory System

for reading as well as for writing. There are three error bits in the APR Status Register which indicate to the software (on an interrupt to the APR) that the ERA Register contains error information. These three bits in the APR Register are: NXM (Bit 25), MB PAR ERR (Bit 27), and SBUS ADR ERR (Bit 29). The software then reads the contents of the ERA Register. After the software reads the ERA Register, it must clear the three bits in the APR Register in order to allow the ERA Register to be ready for a subsequent error.

Organization. Bits 14-35 contain the 22-bit physical address requested by the processor. Bits 0 and 1 of the ERA Register indicate in which of the four words (that are requested of the Memory System) the error occurred. Bit 2 is called the sweep reference bit. It is a 1 if the Memory Error occurred on a reference due to a cache sweep. Bits 4 and 5 comprise a data source code that indicates where data originated. The data may come from memory on a read or a read-pause-write; data may be from the processor and going to memory for a write; data may come from the cache on a page refill or data may come from the cache on a write operation to memory. Bit 6 is the write reference bit. It is a 1 if a memory write operation was in progress when the error occurred. It is a 0 on the read portion of a read-pause-write. The data source field (Bits 4-5) is not valid on an Address Parity Error.

RDERA, Read Error Address Register



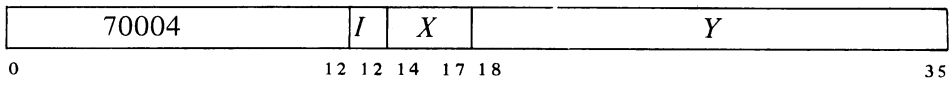
Read the status of the Error Address Register into location *E* as shown.

Bit	*Function*
0-1	Position of the word requested, which was found in error within the four word group.
2	A 1 in this bit indicates that the last memory reference (which was a core memory write back), was made by a cache sweep operation.
3	Not used, reserved by DEC.
4-6	This three bit code consists of two portions. The first, bits 4-5, indicates the data source for the last memory reference. The second, bit 6, indicates whether the memory reference type was read or write.

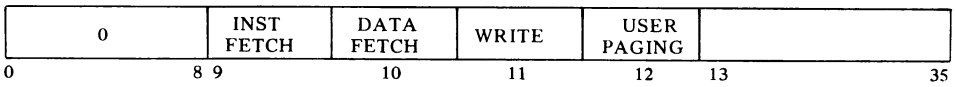
Data SRC Code Bits 4-5	*Write Bit 6*	*Operation*	*Source*
00	0	Processor read or read-pause-write.	Core Memory

10	1	Processor Write	EBox
11	0	Page Refill	Core Memory or Cache
4-5			
11	1	Processor Write	Cache
07-13		Bits 07-08 are indeterminate.	
14-35		Physical address of the first word requested.	

DATAI APR, Data-In Arithmetic Processor



Read the status of the address-break condition into location *E* as shown.



9 Fetch Inst. – Selects access for retrieval of an ordinary instruction. Address by PC.

10 Fetch Data – Selects access for retrieval of an indirect word in an effective address calculation or retrieval of a read only operand (including the object of an XCT instruction).

11 Write – Selects access for writing.

12 User Paging – If User Paging is a 1, the address comparison is enabled for the User Address space. If the User Paging Flag is a 0, the address comparison is enabled for the Executive Address space.

DATAO APR, Data-Out Arithmetic Processor

Set the status of the address-break condition from location E as shown above. Bits 13:35 are the virtual address.

KI10 Processor Conditions

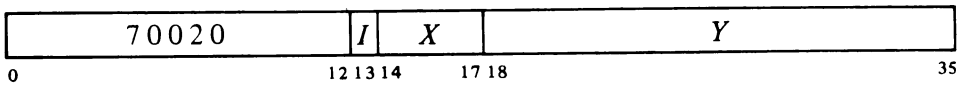
In the KI10, page failures and overflow are handled by trapping, but other internal conditions use the interrupt system. The program can actually assign two channels to the processor – one for error conditions and one specifically for the clock. Control over the Power Failure and Parity Error flags is exercised by a CONO that addresses the priority interrupt system [§2.13]. Control over other conditions and inspection of all are handled by condition IO instructions that address the processor; the CONI also reads some console switches and maintenance functions. The processor also has a data-out instruction through which the program can perform margin checking of the system in both speed and voltage.

The error conditions are generally regarded as important enough to be assigned to the highest priority channel. However for conditions that may be associated with user instructions (a parity error or unanswered memory reference), the common practice is for the error interrupt to switch over to the lowest priority channel by means of a program-set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

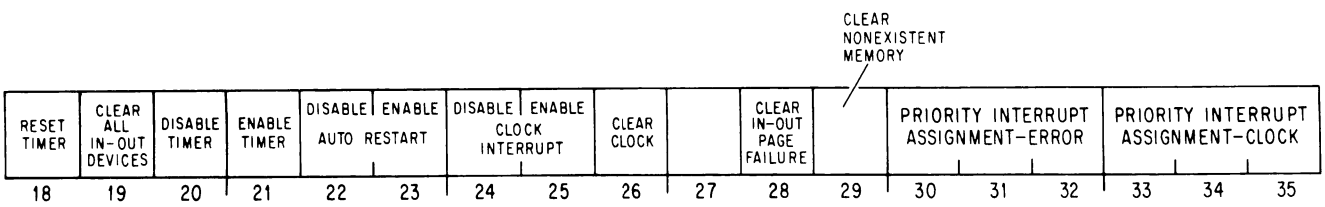
One of the features controlled by the CONO for the processor is the automatic restart after power failure. This restart applies only when the levels on the power mains go below specification while the processor is running, and the power switch is on when power is restored – the machine never begins operation by itself when the operator turns the power switch on or off. Inadequate power, over temperature, etc are indicated by the Power Failure flag. In order for the processor to restart itself, the program must respond in a particular way to the setting of Power Failure. If the program fails to respond properly, there is no restart.

The processor device code is 000, mnemonic APR.

CONO APR, Conditions Out, Arithmetic Processor



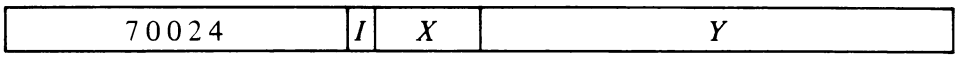
Assign the interrupt channels specified by bits 30–35 of the effective conditions *E* and perform the functions specified by bits 18–29 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



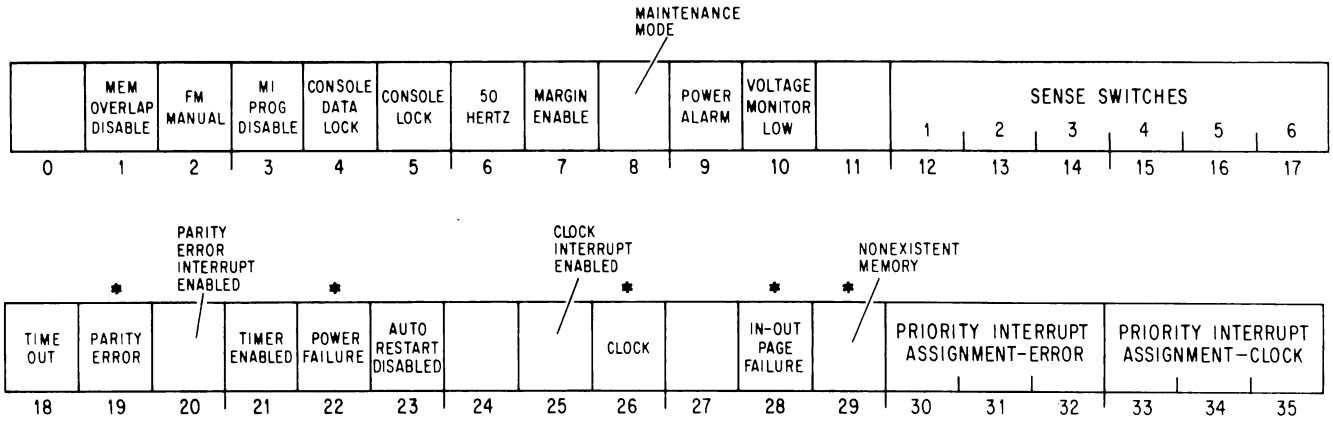
A 1 in bit 19 produces the IO reset signal, which clears the control logic

in all of the peripheral equipment (but affects neither the priority interrupt system nor the processor conditions).

CONI APR, Conditions In, Arithmetic Processor



Read the status of the processor (as well as various console switches and maintenance functions) into location *E* as shown.



Interrupts are requested on the error channel (assigned by bits 30–32 of the CONO) by the setting of Power Failure, In-out Page Failure, Nonexistent Memory, and if enabled, Parity Error. The setting of Clock Flag, if enabled, requests an interrupt on the clock channel (assigned by bits 33–35 of the CONO).

Bits 12–17 reflect the states of the console sense switches, which are specifically for operator communication with the program. Bits 1–5 reflect the settings of various console operating switches; for information on these switches refer to Appendix F1. Bits 7–10 are maintenance functions for which the reader should refer to Chapter 10 of the maintenance manual.

*These bits cause interrupts.

The processor does not actually have a maintenance mode – the bit is simply the OR function of a number of console switches, any one of which being on implies that the processor is being operated for maintenance purposes.

6 The system is operating on 50 Hz line power. This is important to the program, not only because some IO devices run slower on 50 Hz, but because the program must compensate for the time difference when using the line frequency clock (bit 26).

18 Bit 21 is 1 and the program has not reset the timer (CONO APR, bit 18) during the last 1.2 seconds (the period of the timer may vary from 1.2 to 1.5 seconds). The setting of this flag clears the processor and the peripheral equipment, and restarts the processor in kernel mode at location 70.

The timer provides a restart similar to that following power failure. Running the machine under margins may result in significant logical errors. If the timer is enabled, failure of the program to reset it about every second allows it to time out. The restart instruction should set up PC, which would otherwise be clear.

19 A word with even parity has been read from core memory. If bit 20 is 1, the setting of Parity Error requests an interrupt on the error channel [*see cautions below*].

22 Ac power has failed. The program should save PC, the flags, mode information and fast memory in core, and halt the processor. Note that PC may point to an interrupt service routine rather than the main program.

The setting of this flag requests an interrupt on the error channel. After 4 ms the processor is cleared. But at that time, if the power switch is on and the program has cleared Power Failure (CONO PI,400000) and enabled the auto restart (CONO APR, 010000), then when adequate power levels are restored, the processor will resume normal operation by executing the instruction in location 70 in kernel mode.

The restart instruction should set up PC, which would otherwise be clear.

26 This flag is set at the ac power line frequency and can thus be used for low resolution timing (the clock has high long term accuracy). If bit 25 is 1, the setting of the Clock flag requests an interrupt on the clock channel.

An interrupt page failure caused by the console address break switch sets this flag instead of producing an address failure [§2.15].

28 A page failure has occurred in an interrupt instruction. The setting of this flag requests an interrupt on the error channel.

Note: A page failure in an interrupt instruction is regarded as a fatal error, and it causes an interrupt instead of a page failure trap. The kernel mode program is expected to set up the interrupt instructions so that a page failure simply cannot occur.

PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

29 The processor attempted to access a memory that did not respond within 100 μs. The setting of this flag requests an interrupt on the error channel [*see cautions below*].

Programming Cautions. When handling parity error or nonexistent memory interrupts, the programmer should beware of the following.

Remember that during the grant procedure, the interrupt system is otherwise static and the program continues. Moreover the processor is effectively at the far end of the bus.

♦ Should an error flag be set during an interrupt grant, the processor would handle a lower priority interrupt before getting to the processor interrupt. This means PC may be pointing to a lower level interrupt service routine rather than the program level at which the error occurred.

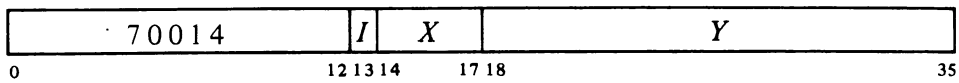
♦ Even without inadvertent interference from another channel, it is quite likely the processor will perform one or perhaps two more instructions between the time the error flag sets and its interrupt starts. Hence even though PC is at the correct program level, it may well be pointing to the first or second instruction following the one in which the error occurred.

♦ A processor error interrupt that switches over to a lower priority channel should not return to the interrupted program, as the error may simply recur, producing a second processor interrupt before the error-handling interrupt for the first. This could happen because PC is actually pointing to the offending instruction, but beyond that, one error often begets another – consider the case of PC counting into a nonexistent memory.

In any event, it is generally not worthwhile to return to any program without first finding out what has gone wrong.

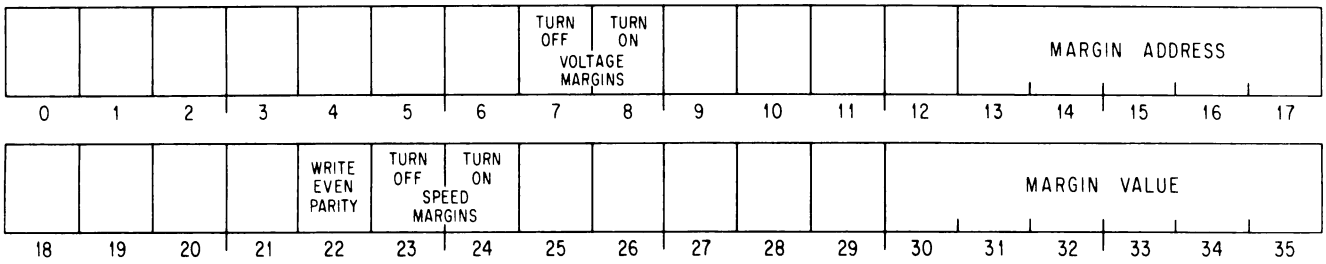
♦ The error may have originated from a console key function, and thus be hidden from any investigation by the program.

DATAO APR, Maintenance Data Out, Arithmetic Processor



This instruction is primarily for maintenance, for which further information is given in Chapter 10 of the KI10 Maintenance Manual.

Supply diagnostic information and perform diagnostic functions according to the contents of location *E* as shown.



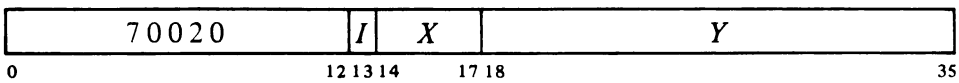
The margin value supplied by bits 30–35 of the output word is translated to a voltage in the range 0–10 volts by a D-A converter, whose output is available at pin 2S02V2. Running margins requires a slowdown capacitor in the converter. But turning off the margin enable switch cuts out the capacitor, making the converter output suitable for external use, such as for operating audio equipment to play Bach or rock or Bacharach.

KA10 Processor Conditions

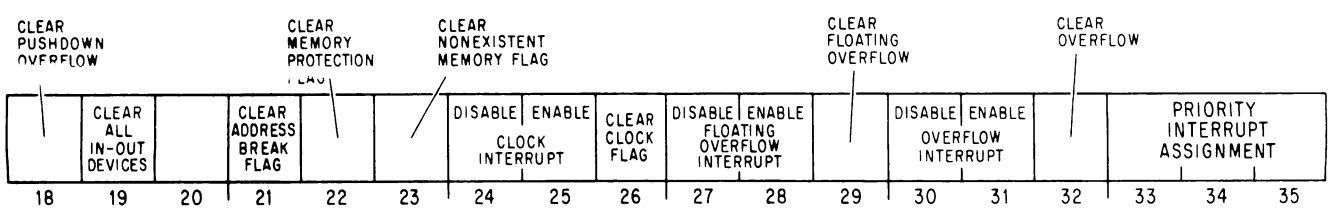
All KA10 processor conditions act through the interrupt on a single channel assigned to the processor. Flags for power failure and parity error are handled by the condition IO instructions that address the priority interrupt system [§2.13]. The remaining flags are handled by condition instructions that address the processor. Its device code is 000, mnemonic APR.

Most of these conditions are generally regarded as important enough to be assigned to the highest priority channel. Except in the case of a power failure however, the common practice is for the processor interrupt to switch over to the lowest priority channel by means of a program-set request. Then the time taken to handle the situation, which may well be considerable, cannot interfere with high priority events.

CONO APR, Conditions Out, Arithmetic Processor



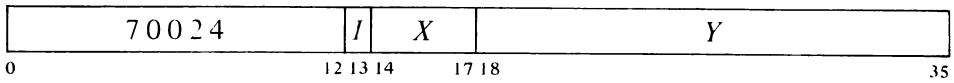
Assign the interrupt channel specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 18–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



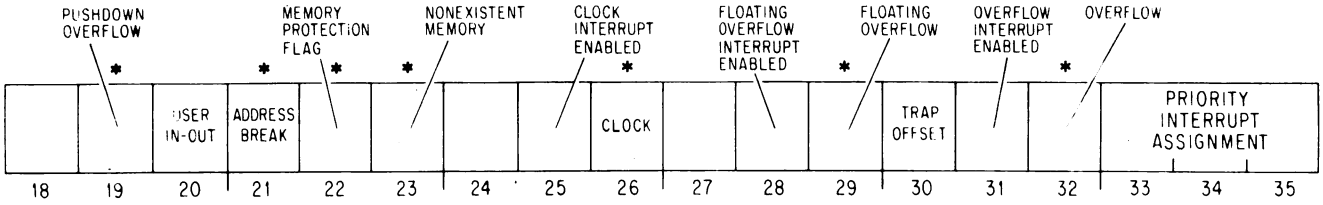
Enabling a particular flag to interrupt means that henceforth the setting of the flag will request an interrupt on the channel assigned (by bits 33–35) to the processor. Disabling prevents the flag from triggering a request.

A 1 in bit 19 produces the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system, nor the processor flags cleared by this instruction or CONO PI,).

CONI APR, Conditions In, Arithmetic Processor



Read the status of the processor into the right half of location E as shown (all interrupt requests are made on the channel assigned to the processor).



*These bits cause interrupts on the channel assigned to the processor, as do Power Failure and Parity Error, bits 18 and 19 read by a CONI PI,.

With the possible exception of an illegal memory reference on an instruction fetch, if the highest priority active channel is assigned to the processor, then the occurrence of any processor interrupt condition is guaranteed to produce a processor interrupt with no lower priority interrupt intervening between it and the program level at which the processor condition occurred. The actual relationship between PC and the instruction associated with a given condition is as stated in its description.

19 Pushdown Overflow – in a PUSH or PUSHJ the count in AC left reached zero; or in a POP or POPJ the count reached −1. The setting of this flag requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred. The location of the offending instruction is implied by PC for PUSH or POP, is indicated by the last item in the stack for PUSHJ, but is indeterminate for POPJ.

20 User In-out – even if the processor is in user mode, there are no instruction restrictions (but memory restrictions still apply) [§2.16].

PC bears no relation to the break if the access was requested for a console key function.

21 Address Break – while the console address break switch was on, the processor requested access to the memory location specified by the address switches and the memory reference was for the purpose selected by the address condition switches as follows:

The instruction switch was on and access was for retrieval of an instruction (including an instruction executed by an XCT or contained in an interrupt location or a trap for an unimplemented operation) or an address word in an effective address calculation.

The data fetch switch was on and access was for retrieval of an operand (other than in an XCT).

The write switch was on and access was for writing a word in memory, other than in a read-modify-write.

The setting of this flag requests an interrupt, at which time PC points to the instruction that was being executed or to the one following it.

22 Memory Protection – a user program attempted to access a memory location outside of its area or to write in a write-protected part of its area and the user instruction was terminated at that time. The setting of this flag requests an interrupt, at which time PC points either to the instruction that caused the violation or to the one following it, unless the illegal reference was for fetching an instruction. In this exceptional case it is possible for a lower level interrupt to occur between the violation and its interrupt, even with the processor assigned to the highest priority active channel.

This flag can also be set by an instruction executed from the console while the USER MODE light is on, in which case PC bears no relation to the violation.

23 Nonexistent Memory – the processor attempted to access a memory that did not respond within 100 μs. The setting of this flag requests an interrupt, at which time PC points either to the instruction containing the unanswered reference or to the one following it.

PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

26 Clock – this flag is set at the ac power line frequency and can thus be used for low resolution timing (the clock has high long term accuracy). If bit 25 is set, the setting of the Clock flag requests an interrupt.

29 Floating Overflow – this is one of the flags saved in a PC word, and the conditions that set it are given at the beginning of §2.9. If bit 28 is set, the setting of Floating Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.

30 Trap Offset – the processor is using locations 140–161 for unimplemented operation traps and interrupt locations.

32 Overflow – this is one of the flags saved in a PC word, and the conditions that set it are given at the beginning of §2.9. If bit 31 is set, the setting of Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.

CAUTION

For an address break, a memory protection violation, a parity error, or a nonexistent memory, a processor error interrupt that switches over to a lower priority channel should not return to the interrupted program, as the processor will fetch the next user instruction before it accepts the program-set interrupt request. This makes it very likely that the same error will recur, producing a loop between the processor interrupt and the interrupted program.

2.15 KL10 PROGRAM AND MEMORY MANAGEMENT

General information about the machine modes and paging procedures is given in Chapter 1; in particular, at the end of the introductory remarks and at the end of 1.3. Here we are concerned principally with the special instructions the Monitor uses to operate the system, the special effects that ordinary instructions have in executive mode, and certain hardware procedures; in particular, paging and page failures, that are necessary for an understanding of executive programming.

User Programming. As far as user programming is concerned, all of the necessary information has already been presented. For convenience however, we list here the rules the user must observe. (Refer to the Monitor manual for further information including use of the Monitor for input-output.)

If an area of memory is write-protected, *eg* for reentrant program shared by several users, do not attempt to store anything in it. In particular do not execute a JSR or JSA into a write-protected page.

Use the MUUO codes 040-077 only in the manner prescribed in the Monitor manual. In general, unless they are prescribed for special circumstances, code 000 and the unassigned codes should not be used.

Do not use HALT (JRST 4), or JEN (JRST 12, [specifically JRST 10,]).

Unless User In-Out is set do not give any IO instruction with device code less than 740. The program can determine if User In-out is set by examining bit 6 of the PC word stored by JSR, JSP or PUSHJ.

If your public program has the use of concealed programs, do not reference a location in a concealed page for any purpose except to fetch an instruction from a valid entry point, *ie* a location containing a JRST 1,.

The user can give a JRSTF (JRST 2,) but a 0 in bit 5 of the PC word does not clear User (a program cannot leave user mode this way); and a 1 in bit 6 does not set User In-out, so the user cannot void any of the instruction restrictions himself. Note that a 0 in bit 6 will clear User In-out, so a user can discard his own special privileges. Similarly, a 1 in bit 7 sets Public, but a 0 does not clear it, so a public program cannot enter concealed mode this way.

The above rules are the result of KL10 hardware characteristics. But in a real sense many of these rules are actually transparent to the user, in particular the whole paging setup is invisible.

Although the hardware allows for user virtual address spaces that are scattered and/or very large (*eg* larger than available physical core), the actual constraints will be dictated by the particular Monitor and the system manager. It may be desirable (for compatible operation with KA10 systems) to enforce a two-segment virtual address space that mimics the one imposed by the KA10 hardware. In any case, the user must write a sensible program, which can be handled easily and cheaply by the system; if he uses addresses a few to a page all over memory, his program can be run but will require a much larger amount of core than necessary or cause excessive page swapping.

Paging. All of memory, both virtual and physical, is divided into pages of 512 words each. The Virtual Memory Space addressable by a program is 512 pages, the locations in Virtual Memory are specified by 18-bit addresses, where the left nine bits (18-26) specify the page number, and the right nine (27-35) the location within the page. Physical Memory can contain 8192 pages and requires 22-bit addresses, where the left thirteen bits (13-26) specify the page number. The hardware maps the Virtual Address Space into a part of the Physical Address Space by transforming the 18-bit addresses into 22-bit addresses. In this mapping, the right nine bits of the Virtual Address are not altered. In other words, a given location in a Virtual Page is the same location in the corresponding Physical Page. The transformation maps a Virtual Page into a Physical Page by substituting a 13-bit Physical Page Number for the 9-bit Virtual Page Number. The mapping procedure is carried out automatically by the hardware, but the page map that supplies the necessary substitutions is set up in core by the Kernel Mode Program. Each word in the map provides information for mapping two consecutive pages with the substitution for the even numbered page in the left half, and the odd numbered page in the right half.

	← EVEN HALFWORD →	← ODD HALFWORD →
0	ENTRY FOR PAGE 0	ENTRY FOR PAGE 1
1	ENTRY FOR PAGE 2	ENTRY FOR PAGE 3
•		
•		
•		
377	ENTRY FOR PAGE 776	ENTRY FOR PAGE 777

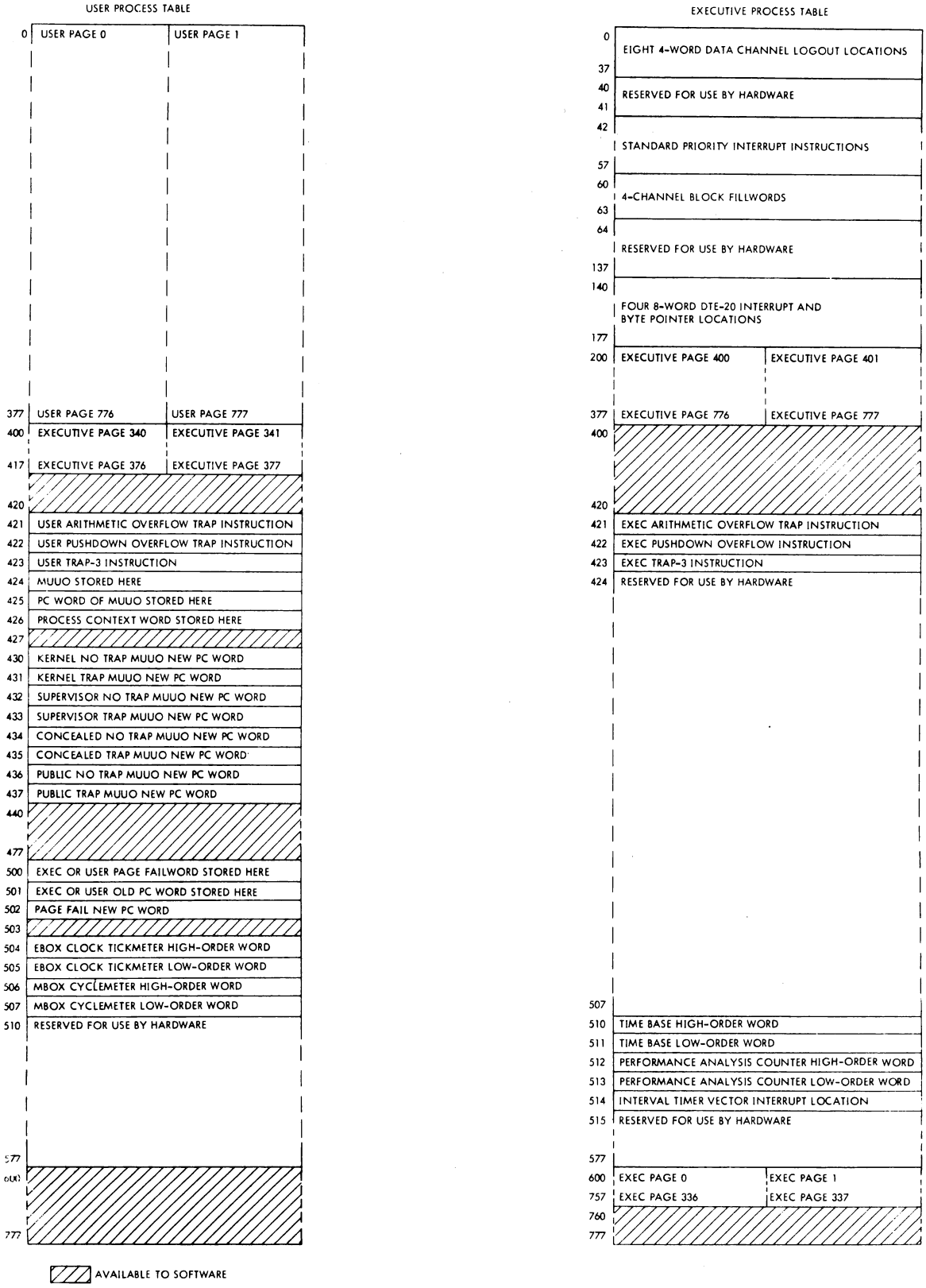
If KL10 microcode paging mode is used, this diagram of the in-core page table format is incorrect. KL10 microcode paging mode is used at both the Stanford AI Lab and the Stanford LOTS computer facility.

PAGE MAP PARTITIONING

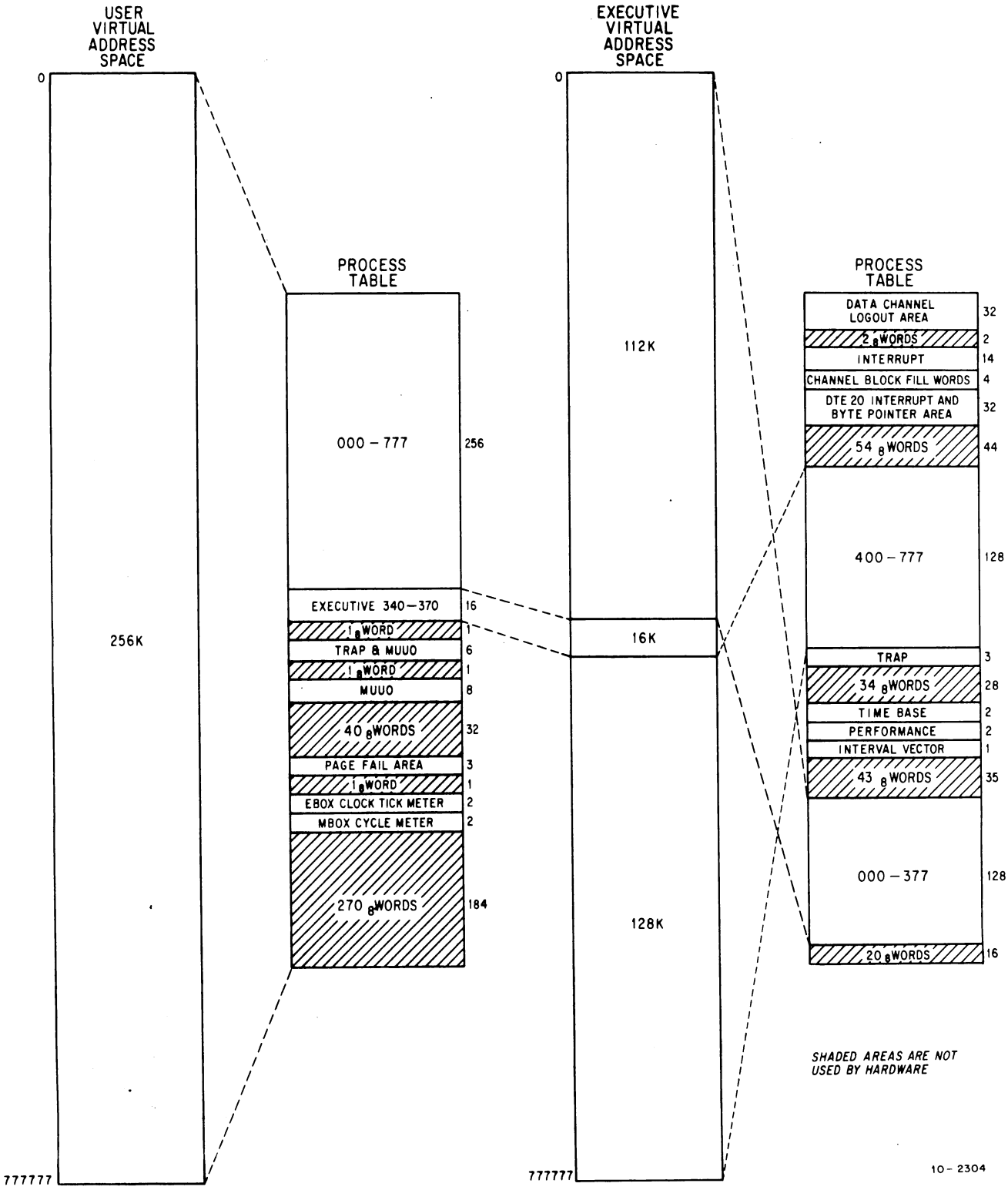
The paging hardware contains two 13-bit registers that the monitor loads to specify the Physical Page Number of the User and Executive Process Tables. These registers are the User Base Register (UBR) and the Executive Base Register (EBR) respectively. To retrieve a map word from a process table, the hardware uses the appropriate base page number as the left thirteen bits of the Physical Address and some function of the Virtual Page Number as the right nine bits. Since there are 512 pages in a Virtual Address space, and the mapping for two pages is stored in one word, 256 words (1/2 page) is required to map an address space. The user space is mapped by locations 0-377 of the User Process Table. The executive space is mapped in three pieces: Executive pages 00-337 are mapped by 600-757 of the Executive Process Table, pages 340-377 are mapped by 400-417 of the User Process Table, and pages 400-777 are mapped by 200-377 of the Executive Process Table.

To find the desired substitution from the 9-bit Virtual Page Number, the hardware uses bits 18-25 to address the location and bit 26 to select the half word (0 for left, 1 for right). The entire Executive Virtual Address space is

mapped. This is different from the KI10, which does not map pages 0-337. The following illustrations show the organization of the Virtual Address spaces, the Process Tables, and the mappings for both User and Executive. The first illustration gives the correspondence between the various parts of each address space and the corresponding parts of the Process Table for it. The second illustration lists the detailed configuration of the Process Tables. Any table locations not used by the hardware can be used by the monitor for software functions. Note that the numbers in the half locations in the page map are the Virtual Pages for which half words give the physical substitutions. Hence, location 217 in the User Page Map contains the Physical Page Numbers for Virtual Pages 436 and 437.

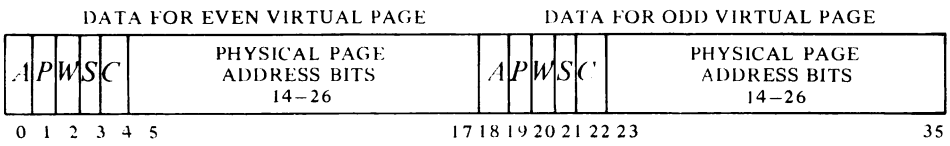


PROCESS TABLE CONFIGURATION KL10



VIRTUAL ADDRESS SPACE AND PAGE MAP LAYOUT KL10

In addition to address mapping information, each Page Table entry contains four bits which determine whether the page is accessible, Public, and/or writable, and whether data from it may be cached. Finally, one bit in the entry (S-bit), is available for use by software. Each word in the Page Map has this format to supply the necessary information for two Virtual Pages.

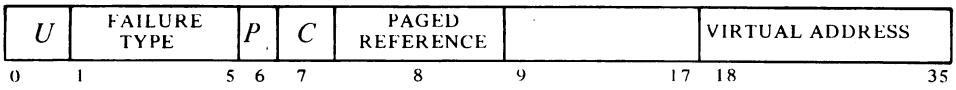


Bits 5-17 and 23-35 contain the Physical Page Numbers for the even and odd numbered Virtual Pages corresponding to the map location that holds the word. The properties represented by 1s in the remaining bits are as follows.

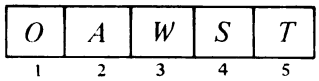
Bit	*Meaning of a 1 in the Bit*
A	Access Allowed
P	Public
W	Writable (Not write-protected)
S	Software (Not interpreted by hardware)
C	The cache is enabled for references to this page.

Page Failure. In all cases except in-out page failure, if the paging hardware cannot make the desired reference, it terminates the instruction immediately without disturbing memory, the accumulators or PC, and causes a Page Fail trap. The Page Fail word is placed in location 500 of the User Process Table. The flags and PC word are placed in location 501 of the same table. Then processor sets up the flags and PC according to the new PC word in location 502 of the User Process Table and restarts the processor at the location then addressed by PC.

The Page Fail word supplies this information.



Bit 0(U) is a zero if the violation occurred on an attempted Executive Virtual reference. Bit 0(U) is a one if the attempted reference was to a User Virtual Address, whether by a User Program or by the executive using PXCT.



The effect of this bit depends on several factors. The cache must be turned on by bits 18 and 19 of CONO PAG for this to have any effect. What is more, this bit never prevents the processor from finding data in the cache if it was brought in by another reference. Its only effect is to allow the processor to bring data into the cache while satisfying a reference to this page.

If bit 1 is a zero, bits 2-5 have the format A, W, S and T, and are from the page map entry for the Virtual Page specified by bits 0 and bits 18-26. T is a zero if this reference was a read, but is a one if this reference was a write, read-write, or a read-modify-write.

If bit 1 is a one, bits 1-5 contain a code indicating the type of failure as follows:

37 ARX DATA PARITY – The processor detected bad parity on a word read from memory or the cache into ARX.

36 AR DATA PARITY – The processor detected bad parity on a word read from memory or the cache into AR.

25 PAGE TABLE PARITY – The processor detected bad parity for a hardware page table entry.

23 ADDRESS BREAK – An attempted paged memory reference matched an address specified as the address break address, and the reference was of a type enabled in the most recent DATAO APR. (For the types see 2.14 DATAO APR.)

22 PAGE REFILL FAILURE – This is a hardware malfunction. The paging hardware did not find the mapping information for some Virtual Page in the hardware page table so it loaded paging information from the page map into the table but still could not find it.

21 PROPRIETARY VIOLATION – An instruction in a public page has attempted to reference a concealed page or transfer control into a concealed page at an invalid entry point (one not containing a JRST1).

Monitor Programming. The Kernel Mode program is responsible for the overall control of the system. It is the only program that has no instruction restrictions. The Kernel program handles all in-out for the system and must set up the page maps, trap locations, interrupt locations, and the like. The supervisor program labors under the same instruction restrictions as the user but has no way of bypassing them – they always apply. The Kernel program supplies data tables of all kinds to the supervisor program, and the latter cannot affect them. The supervisor can give a JRSTF that clears Public provided it is also setting User; in other words, the supervisor can return control to a concealed program but cannot enter Kernel Mode by manipulating the flags. The PC words supplied by MUUOs can manipulate the flags in any way, switching arbitrarily from one mode to another, but these are in the process table and assumed to be under control solely of Kernel Mode.

The Kernel Mode program allocates blocks of Fast Memory. While a User Program is running AC, Index, and Fast Memory references are made to the

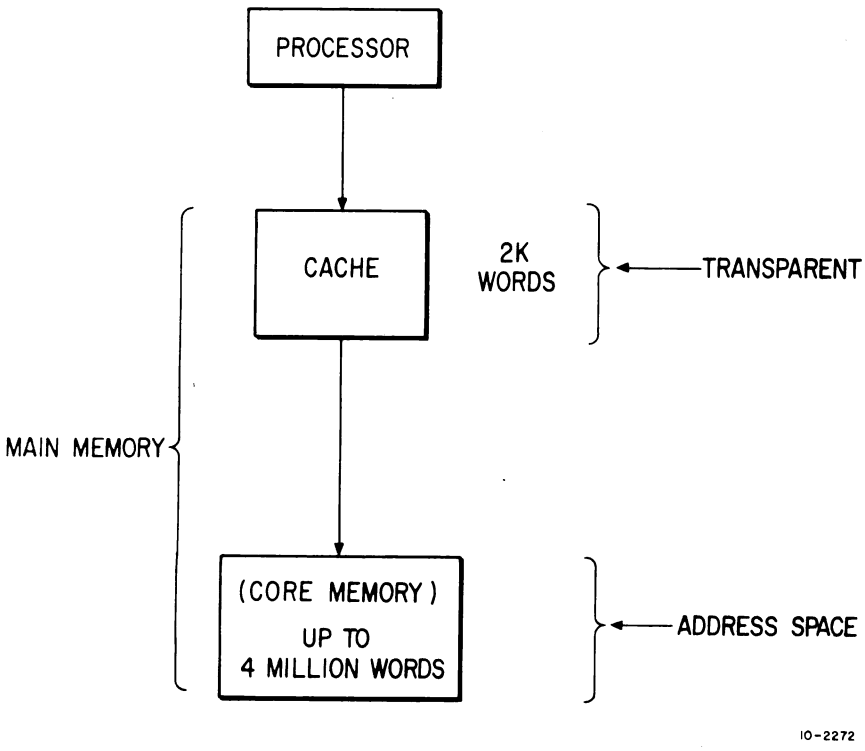
block selected as Current. When the User calls the Monitor (via all MUUO), the Monitor will ordinarily re-select AC blocks so the User's ACs are the "Previous Block" and thus accessible by PXCT, and a different block is the "Current Block" for use by ordinary AC, Index, and Fast Memory references. Thus, the Monitor will not save and restore a User's AC block except when switching users and can avoid even this for a few high priority users or interrupt routines by assigning them unique blocks. Block 7 is reserved for use by the micro code.

If the Monitor shared block 0 with any users, it would have to store the user accumulators even when taking control only temporarily.

Even while User is set, the interrupt instructions are not part of the user program and are thus subject only to executive restrictions. As interrupt instructions, JSR, JSP, and PUSHJ automatically take the processor out of user mode to jump to an executive service routine. An MUUO can also be used. The paging hardware has one non-IO instruction and two condition IO instructions primarily for diagnostic purposes. Otherwise control over the system is exercised by data IO instructions. The device code for the paging hardware is 010, mnemonic PAG.

The trap instructions are executed in the user address space if caused by the user.

Cache Memory. The KL10 cache is a high speed automatic Buffer Memory that is generally transparent to the timesharing user and which holds some selection of words from the Main Memory System for the purposes of reducing access time and reducing the percentage of available Main Memory Cycles needed by the Central Processor.



SIMPLIFIED CACHE BLOCK

Organization. The structure consists of four blocks, each holding 128 4-word groups. Each group contains words whose addresses differ only in the two least significant bits. In addition, each word of the group has two extra bits that indicate whether this word is valid in the cache, and whether the word has been written by the processor; hence, leaving Main Memory currently in error. The 128 groups in each block are addressed by bits 27-33 of the address supplied by the processor. This means that selection of one of the 512 words within a block can be made using that portion of the address not subject to paging. In addition, each group has a 13-bit address register which holds the Physical Page Number of the four words within that group. Also, cache use logic is incorporated to keep track of the order in which the four cache blocks of a given group are used since there are four cache blocks, each consisting of 128 4-word groups. 128 entries must be maintained to keep track of the order in which all cache blocks in the cache are used. Consequently, a Use Table containing 128 locations is employed by the use logic to maintain the use history of the cache.

Processor Requests. On any processor request, the high-order nine bits of the Virtual Address select a Hardware Page Table entry which, if valid, provides a 13-bit Physical Page Number and access control bits. If the hardware page table has no valid entry for the requested page, a new request is generated to get it, and the original request is restarted.

The low-order nine bits of the Virtual Address are simultaneously presented to each of the four cache blocks, selecting from each block one word, its valid and written bits, and its physical group address.

The four group addresses so obtained are compared with the Physical Page Number from the Hardware Page Table, resulting in either one match or none. If there is no valid match, the Use Table is inspected to determine which block was least recently used. If any words in that block are written, a write back cycle is initiated to update core.

Processor Reads. On read cycles, if there was no match, or the selected valid bit was off, a main memory read is initiated to fill in the selected group.

Processor Writes. On write cycles, the data from the processor is written into the cache. Main Memory is not updated until either a cache sweep or another processor request forces a write back.

Channel Reads. Words read from Core Memory and going to a device behave as a cache read for the Physical Address supplied by the channel. Here words are handled by the channel in four word groups. If however, words are not found, or are not all valid, a core cycle is requested and the cache is not disturbed. Any words that are missing from the cache group are filled in by words fetched from Core Memory.

Channel Writes. Words transferred from a device and written into Core Memory are checked against the cache. If not found at all, they go directly to Core Memory. If found in the cache, those words in the group which are actually being written are invalidated by setting their valid bits to 0. This is done independently of the written bits, and now the new words are written into Core Memory. Thus, as with any core write cycles, this implies the ability to modify any subset of a group.

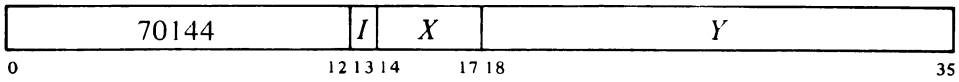
Programming. In any single processor KL10 system, with internal or external memory, but with no devices connected via DF10 like channels to memory, minimal extra programming is necessary for the cache. After power up, the cache must be cleared. This is accomplished by the processor using the SWPIA Instruction, which has been provided for just such a purpose. When the processor is running, power failure detection causes an interrupt on the channel assigned to the processor, and the Monitor, utilizing the SWPUA Instruction, must cause any words with "written" bits equal to 1 to be swept into Core Memory. If multi-port operations are attempted using main memories, special considerations must be made.

Cache Sweep and Validate Main Memory. The cache and the main memory may have different contents. This can be caused by program execution leaving newer data in the cache, an input operation from a peripheral device leaving newer data in main memory, or the cache contents being indetermined when the power is turned on. If the newer data is in the cache and must be written onto a peripheral device, the monitor does a cache sweep on one or more pages to validate main memory by writing the main memory with the newer data from the cache.

If the newer data is in main memory as the result of an input operation or power was turned on in the system, the cache must be invalidated in order that program references will be fetched from main memory the first time.

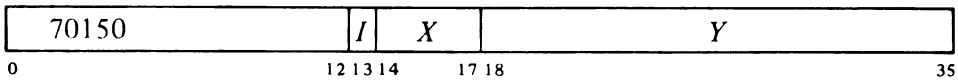
A single page can have both some data correct in the cache and some data correct in Main Memory. Unloading the cache, simultaneously validating main memory, and invalidating the cache would cause the correct cache data to be written into main memory and all the cache locations for that data would be marked invalid, forcing main memory fetches the first time the locations are referenced. Unloading the cache is the normal monitor operation for pages following input operations from devices interfaced directly to external memories such as the MG10.

SWPIA Sweep Cache Invalidate all Pages



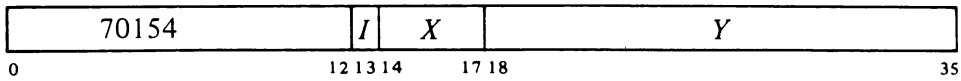
Sweep the entire cache. Clear the valid and written bits for all words in the cache. When the sweep operation has completed, set Sweep Done, requesting an interrupt on the channel assigned to the processor.

SWPVA Sweep Cache Validate all of Main Memory



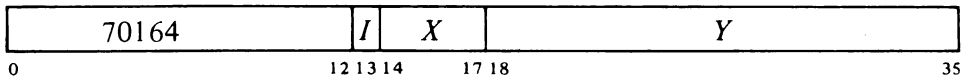
Sweep the entire cache. Write back to main memory all words found in the cache whose written bits are set. Clear the written bits associated with those same entries. The valid entries in the cache are left unchanged when the sweep operation has completed, set Sweep Done, requesting an interrupt on the channel assigned to the processor.

SWPUA Sweep Cache Unloading all Pages-Invalidating Cache



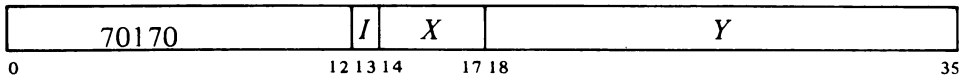
Sweep the entire cache. Write all words found in the cache with their associated written bits equal to 1 into core memory. Clear the written bits and the valid bits associated with all entries in the cache. When the sweep operation has completed, set Sweep Done, requesting an interrupt on the channel assigned to the processor.

SWPIO Sweep Cache Invalidate One Page



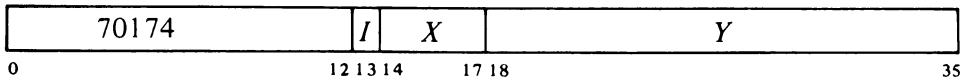
Sweep the physical page in the cache specified by bits 23-35 of *E*. Clear the valid and written bits for all words in the specified page. When the sweep operation has completed, set Sweep Done requesting an interrupt on the channel assigned to the processor.

SWPVO Sweep Cache Validating One Page in Core



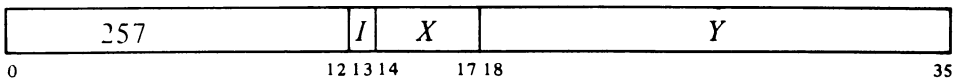
Sweep the physical page in the cache specified by bits 23-35 of *E*. Write all words found in the cache for that page with their associated written bits equal to 1 into core memory. Clear the written bits associated with those same entries. The valid entries in the cache for that page are left unchanged. When the sweep operation has completed, set Sweep Done, requesting an interrupt on the Channel assigned to the processor.

SWPUO Sweep Cache Unloading One Page



Sweep the Physical Page in the cache specified by bits 23-35 of *E*. Write back to Main Memory all words found in the cache for that page, with their associated written bits set. Clear the written bits associated with those same entries. The valid entries in the cache, for that page, are left unchanged. When the sweep operation has completed, set Sweep Done, requesting an interrupt on the channel assigned to the processor.

MAP Map an Address

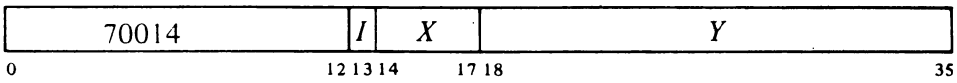


Map the Virtual Address *E* and place the resulting map data in AC as shown.

Bit	*Function*
0	User Address Space
1	If bit 1 is 0 then . . . .
2	Access
3	Writable
4	Software
5	Type
6	Public
7	Cache
8	Paged Ref.
9-13	0
14-35	Physical Address

Bit 0 will be a zero unless the map is performed under PXCT, in which case bit 0 is a copy of previous context user which is user IOT.

DATAO PAG, DATA-OUT PAGING



Set up the paging hardware according to the contents of location *E* as shown.

The following is a list of the bit assignments for the DATAO Pag.

Bit	*Function*
0	Load AC Blocks
1	MBZ
2	Load UBR Invalidate all entries in the Hardware page table
3-5	MBZ
6-8	Current Fast Memory Block 9-11 Previous Fast Memory Block
12-17	MBZ
18	Inhibit Storing Accounting Meters
19-23	MBZ
24-35	User Base

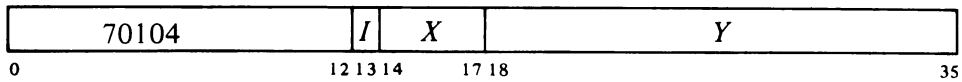
User Base

Bits 1 and 2 are change bits. If bit 0 is 0, ignore bits 6-11. But if bit 0 is 1, select the Fast Memory Block specified in bits 6-8, as the current context AC block, and select the Fast Memory Block specified by bits 9-11 as the previous context AC block.

There are four possible situations as a function of the states of bits 2 and 18.

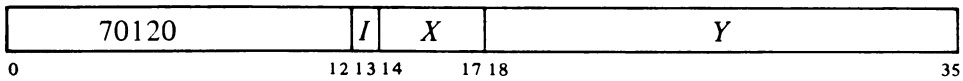
Bit 2	*Bit 18*	*Meaning*
1	0	Update the double length EBOX Meter and double length Cache Meter contained in User Process Table locations 504-507. In each case, read and clear the appropriate 16 bit counter value from the system meter, and add this to the double length meter. The format used is that for double length fixed point numbers where the magnitude is the 70-bit string in bits 1-35 of the high-and low-order words. Bit 0 of the high-order word is the sign, bit 0 of the low-order word is 0. Bits 24-35 are not used in the calculations and are reserved by DEC for future use by the hardware. Now invalidate all entries in the Hardware Page Table and load bits 23-35 into the User Base Register to select the User Process Table.
1	1	Invalidate all entries in the Hardware Page Table and load bits 23-35 into the User Base Register to select the User Process Table.
0	X	If bit 2 is 0, ignore the right half word.

DATAI PAG, Data in Paging



Read the status of the paging hardware into location *E*. Information read is the same as that supplied by the DATAO (bits 0-2 are 1s and bit 18 is 0).

CONO PAG, Conditions Out Paging



Invalidate all entries in the Hardware Page Table. Select the cache strategy specified in bits 18 and 19, traps, and paging enabled as specified by bit 22, and load the Executive Base Register from bits 23-35 of the effective conditions *E* as shown.

The following is a list of the bit assignments for the CONO Pag.

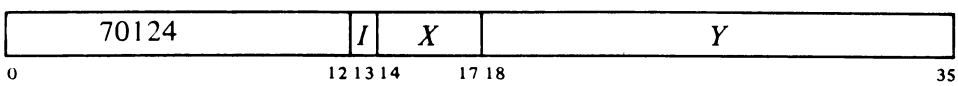
Bit	*Function*
18	Cache Look
19	Cache Load
20	
21	KL10 microcode paging mode.
22	Trap Enable
23-35	Executive Base Register

A 0 in bit 22 disables both traps and paging. In other words, an executive mode program automatically runs in Kernel Mode with all access in the first 256K of Physical Memory unpaged. The cache strategy bits look and load specify how the cache will be used during memory references. There are four possible situations as a function of these two bits as follows:

Bit 18	*Bit 19*	*Page Table C Bit*	
1	1	1	If the look bit and the load bit are both 1, and the C bit in the Page Table is also a 1, the cache is enabled for ALL references to this page.
0	0	X	If both the look bit and the load bit are 0, the cache is disabled for all references and the C bit in the Process Table is ignored.
1	0	X	
	OR		
1	X	0	If the look bit is 1 and the load bit is or, the C bit in the Process Table is 0, references will be cached if, and only if, the word addressed is already in the cache. Otherwise references are from Core Memory and are not placed in the cache.
0	1	X	Do not use this combination.

A 1 in bit 21 invalidates everything this manual says about in-core page table formats.

CONI PAG, Conditions-In Paging



Read the cache strategy, trap enable, and Executive Base Register into the right half of location *E* as shown.

The following is a list of the bit assignments for the CONI PAG.

Bit	*Function*
18	Cache Look
19	Cache Load
20	
21	KL10 microcode paging mode
22	Trap Enable
23-35	Executive Base Register

2.16 KI10 PROGRAM AND MEMORY MANAGEMENT

General information about the machine modes and paging procedures is given in Chapter 1, in particular at the end of the introductory remarks and at the end of §1.3. Here we are concerned principally with the special instructions the Monitor uses to operate the system, the special effects that ordinary instructions have in executive mode, and certain hardware procedures, in particular paging and page failures, that are necessary for an understanding of executive programming.

User Programming. As far as user programming is concerned, all of the necessary information has already been presented. For convenience however we list here the rules the user must observe. [*Refer to the Monitor manual for further information including use of the Monitor for input-output.*]

- If possible, limit your memory needs to 32K, using addresses 0–37777 and 400000–437777, to gain the savings afforded by having the status of a "small user". There are no restrictions of any kind on addresses 0–17 as these are in fast memory and are available to all users (even though page 0 may otherwise be inaccessible).
- If an area of memory is write-protected, *eg* for a reentrant program shared by several users, do not attempt to store anything in it. In particular do not execute a JSR or JSA into a write-protected page.
- Use the MUUO codes 040–077 only in the manner prescribed in the Monitor manual. In general, unless they are prescribed for special circumstances, code 000 and the unassigned codes should not be used.
- Do not use HALT (JRST 4,) or JEN (JRST 12, (specifically JRST 10,)).
- Unless User In-out is set do not give any IO instruction with device code less than 740. The program can determine if User In-out is set by examining bit 6 of the PC word stored by JSR, JSP or PUSHJ.
- If your public program has the use of concealed programs, do not reference a location in a concealed page for any purpose except to fetch an instruction from a valid entry point, *ie* a location containing a JRST 1,.

The user can give a JRSTF (JRST 2,) but a 0 in bit 5 of the PC word does not clear User (a program cannot leave user mode this way); and a 1 in bit 6 does not set User In-out, so the user cannot void any of the instruction restrictions himself. Note that a 0 in bit 6 will clear User In-out, so a user can discard his own special privileges. Similarly a 1 in bit 7 sets Public, but a 0 does not clear it, so a public program cannot enter concealed mode this way.

The above rules are the result of KI10 hardware characteristics. But in a real sense many of these rules are actually transparent to the user, in particular the whole paging setup is invisible. Although the hardware allows for user virtual address spaces that are scattered and/or very large (*eg* larger than available physical core), the actual constraints will be dictated by the particular Monitor and the system manager. It may be desirable (for compatible operation with KA10 systems) to enforce a two-segment virtual address space that mimics the one imposed by the KA10 hardware. In any case the user must write a sensible program, which can be handled easily and cheaply by the system; if he uses addresses a few to a page all over memory, his program can be run but will require a much larger amount of core than necessary or cause excessive page swapping.

Paging

All of memory both virtual and physical is divided into pages of 512 words each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18-bit addresses, where the left nine bits specify the page number and the right nine the location within the page. Physical memory can contain 8192 pages and requires 22-bit addresses, where the left thirteen bits specify the page number. The hardware maps the virtual address space into a part of the physical address space by transforming the 18-bit addresses into 22-bit addresses. In this mapping the right nine bits of the virtual address are not altered; in other words a given location in a virtual page is the same location in the corresponding physical page. The transformation maps a virtual page into a physical page by substituting a 13-bit physical page number for the 9-bit virtual page number. The mapping procedure is carried out automatically by the hardware, but the page map that supplies the necessary substitutions is set up by the kernel mode program. Each word in the map provides information for mapping two consecutive pages with the substitution for the even numbered page in the left half, the odd numbered page in the right half.

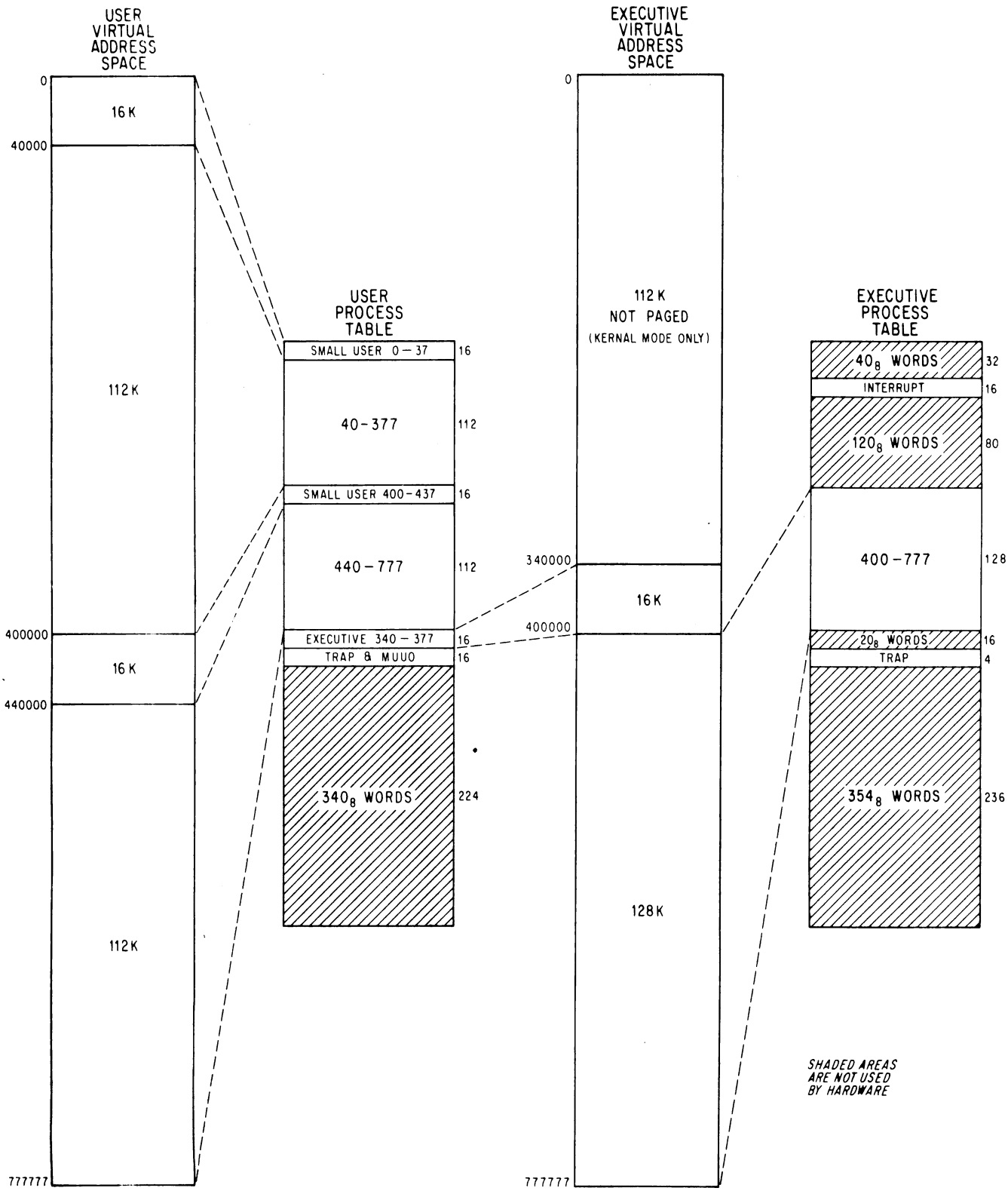
Actually page 0 has only 496 locations using addresses 20–777, as addresses 0–17 reference fast memory, which is unrestricted and available to all programs. (In general a user cannot reference the first sixteen core locations in his virtual page 0.) Throughout this discussion it is assumed that all references are to core and are not made by an instruction executed by an executive XCT [*see below*].

The paging hardware contains two 13-bit registers that the Monitor loads to specify the physical page numbers of the user and executive process tables. To retrieve a map word from a process table, the hardware uses the appropriate base page number as the left thirteen bits of the physical address and some function of the virtual page number as the right nine bits. *Eg* the entire user space of 512 virtual pages at two mappings per word requires a page map of just half a page, and this is the first half page in the user process table. Thus locations 0–377 in the table hold the mappings for pages 0 and 1 to 776 and 777. To find the desired substitution from the 9-bit virtual page number, the hardware uses the left eight bits to address the location and the right bit to select the half word (0 for left, 1 for right). If the Monitor specifies a program as being a small user, that program is limited to two 16K blocks with addresses 0–37777 and 400000–437777. This is pages 0–37 and 400–437, and the mappings are in locations 0–17 and 200–217 in the page map.

The executive virtual address space is also 256K but the first 112K are not paged – in other words any address under 340000 given in kernel mode addresses one of the first 112K locations in physical memory directly. The other 144K is paged for supervisor or kernel mode anywhere into physical memory. For this there are two maps. The map for the second half of the virtual address space uses the same locations in the executive process table as are used in the user process table for the user map (locations 200–377 for pages 400–777). The map for the remaining 16K in the first half of the executive virtual address space is in the *user* process table, the mappings for pages 340–377 being in locations 400–417. Thus the Monitor can assign a different set of thirty-two physical pages (the per-process area) for its own use relative to each user.

Thus when switching from one user to another, the Monitor need change only the user process table. This single substitution can make whatever change is necessary in the executive address space for a particular user.

The illustrations on the next two pages show the organization of the virtual address spaces, the process tables and the mappings for both user and executive. The first illustration gives the correspondence between the various parts of each address space and the corresponding parts of the page



VIRTUAL ADDRESS SPACE AND PAGE MAP LAYOUT

USER PROCESS TABLE

0	USER PAGE 0	USER PAGE 1
17	USER PAGE 36	USER PAGE 37
20	USER PAGE 40	USER PAGE 41
	AVAILABLE TO SOFTWARE IF SMALL USER	
177	USER PAGE 376	USER PAGE 377
200	USER PAGE 400	USER PAGE 401
217	USER PAGE 436	USER PAGE 437
220	USER PAGE 440	USER PAGE 441
	AVAILABLE TO SOFTWARE IF SMALL USER	
377	USER PAGE 776	USER PAGE 777
400	EXECUTIVE PAGE 340	EXECUTIVE PAGE 341
417	EXECUTIVE PAGE 376	EXECUTIVE PAGE 377
420	USER PAGE FAILURE TRAP INSTRUCTION	
421	USER ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	USER PUSHDOWN OVERFLOW TRAP INSTRUCTION	
423	USER TRAP 3 TRAP INSTRUCTION	
424	MUUO STORED HERE	
425	PC WORD OF MUUO STORED HERE	
426	EXECUTIVE PAGE FAILURE WORD	
427	USER PAGE FAILURE WORD	
430	KERNEL NO TRAP NEW MUUO PC WORD	
431	KERNEL TRAP NEW MUUO PC WORD	
432	SUPERVISOR NO TRAP NEW MUUO PC WORD	
433	SUPERVISOR TRAP NEW MUUO PC WORD	
434	CONCEALED NO TRAP NEW MUUO PC WORD	
435	CONCEALED TRAP NEW MUUO PC WORD	
436	PUBLIC NO TRAP NEW MUUO PC WORD	
437	PUBLIC TRAP NEW MUUO PC WORD	
440		
	AVAILABLE TO SOFTWARE	
777		

EXECUTIVE PROCESS TABLE

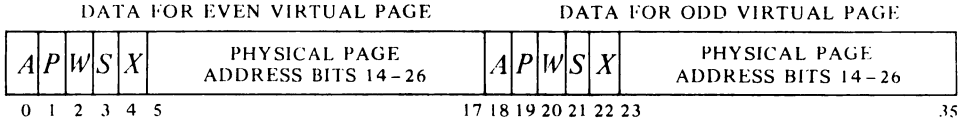
0		
	AVAILABLE TO SOFTWARE	
37		
40	EXECUTIVE LUUO STORED HERE	
41	LUUO HANDLER INSTRUCTION	
42		
	STANDARD PRIORITY INTERRUPT INSTRUCTIONS	
57		
60		
	AVAILABLE TO SOFTWARE	
177		
200	EXECUTIVE PAGE 400	EXECUTIVE PAGE 401
377	EXECUTIVE PAGE 776	EXECUTIVE PAGE 777
400		
	AVAILABLE TO SOFTWARE	
417		
420	EXECUTIVE PAGE FAILURE TRAP INSTRUCTION	
421	EXECUTIVE ARITHMETIC OVERFLOW TRAP INSTRUCTION	
422	EXECUTIVE PUSHDOWN OVERFLOW TRAP INSTRUCTION	
423	EXECUTIVE TRAP 3 TRAP INSTRUCTION	
424		
	AVAILABLE TO SOFTWARE	
777		

PROCESS TABLE CONFIGURATION

map for it. The second illustration lists the detailed configuration of the process tables. Any table locations not used by the hardware can be used by the Monitor for software functions. Note that the numbers in the half locations in the page map are the virtual pages for which the half words give the physical substitutions. Hence location 217 in the user page map contains the physical page numbers for virtual pages 436 and 437.

There is no requirement that the accessible space be continuous – it can be scattered pages. The convention however is for the accessible space to be in two continuous virtual areas, low and high, beginning respectively at locations 0 and 400000. The low part is generally unique to a given user and can be used in any way he wishes. The (perhaps null) high part is a reentrant area, which is shared by several users and is therefore write-protected. The small user configuration is consistent with this arrangement.

Although the virtual space is always 256K by virtue of the addressing capability of the instruction format, the Monitor usually limits the actual address space for a given program by defining only certain pages as accessible. The Monitor also specifies whether each page is public or not and writeable or not. Each word in the page map has this format to supply the necessary information for two virtual pages.



Bits 5–17 and 23–35 contain the physical page numbers for the even and odd numbered virtual pages corresponding to the map location that holds the word. The properties represented by 1s in the remaining bits are as follows.

Bit	*Meaning of a 1 in the Bit*
A	Access allowed
P	Public
W	Writeable (not write-protected)
S	Software (not interpreted by the hardware)
X	Reserved for future use by DEC (do not use)

Associative Memory. If the complete mapping procedure described above were actually carried out in every instance, the processor would require two memory references for every reference by the program. To avoid this the paging hardware contains a 32-word associative memory, in which it keeps the more recently used mappings for both the executive and the current user. Each word is divided into two parts with one part containing a virtual page number specified by the program and the other containing the corresponding physical page number as determined from the page map. Hence the associative memory is a page table made up of a list of virtual pages and a list of physical pages, each with thirty-two corresponding locations. In the virtual list, each entry contains a 9-bit virtual page number, a single bit that indicates whether the specified page is in the user or executive address space, and a bit that indicates whether the entry is valid or not (it is not suitable to clear a location as 0 is a perfectly valid page number). Each corresponding entry in the physical list contains a 13-bit physical page number and the *P*, *W* and *S* bits from the map half word for that page. The *A* bit is not needed in the table as the mapping is not entered into the table at all if the page is not accessible.

The program can inspect the contents of the page table by using the **MAP** instruction and IO instructions that address the paging hardware [*see below*].

At each reference the hardware compares the page number supplied by

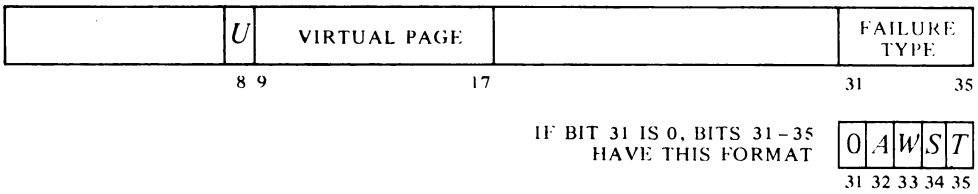
the program with those in the virtual part of the page table. If there is a match for the appropriate address space, the corresponding entry in the physical list is used as the left thirteen bits in the physical address (provided of course that the reference is allowable according to the *P* and *W* bits). If there is no match, the hardware makes a memory reference to get the necessary information from the page map and enters it into the page table at the location specified by a reload counter. This counter is incremented whenever it is used to reload the table, and also whenever the location to which it points is used for a mapping. Hence the counter tends to stay away from locations containing the page numbers most frequently referenced.

This memory reference is referred to as a "page refill cycle."

Page Failure

A page failure that occurs during an interrupt instruction terminates the instruction and sets the In-out Page Failure flag, requesting an interrupt on the error channel assigned to the processor. In all other circumstances, if the paging hardware cannot make the desired memory reference, it terminates the instruction immediately without disturbing memory, the accumulators or PC, places a page fail word in the user process table, and causes a page failure trap. If the attempted reference is in user virtual address space, the page fail word is placed in location 427 of the user process table, and the processor executes the trap instruction in location 420 of the same table. If the attempted reference is in executive virtual address space, the page fail word is placed in location 426 of the *user* process table, and the processor executes the trap instruction in location 420 of the *executive* process table. The trap instruction is executed in the same address space in which the failure occurred. The page fail word supplies this information.

When a page failure trap instruction is performed, PC points to the instruction that failed (or to an XCT that executed it), unless the failure occurred in an overflow trap instruction, in which case PC points to the instruction that overflowed. After taking care of the failure, the processor can always return to the interrupted instruction. Either the instruction did not change anything, or the failure was in the second part of a two-part instruction, where First Part Done being set prevents the processor from repeating any unwanted operations in the first part.



Whether the violation occurred in user or executive virtual address space is indicated by a 1 or a 0 in bit 8. If bit 31 is 1, the number in bits 31-35 ($\geq$ 20) indicates the type of "hard" failure as follows.

23 Address failure – this is a simulated page failure caused by the satisfaction of an address condition selected from the console. It indicates that while the console address break switch was on and the Address Failure Inhibit flag was clear (bit 8 of the PC word), the processor initiated a page check for access to the memory location that was specified by the paging and address switches and for which a comparison was enabled, and the intended memory reference was for the purpose selected by the address condition switches as follows:

The instruction fetch switch was on and the requested access was for retrieval of an ordinary instruction, including an instruction executed by an XCT or an LUUO (address 41).

Since a user page failure trap instruction is executed in user address space, the Monitor should be careful not to have the trap instruction do indirect addressing that might cause another page failure.

Whether or not a comparison can be made is a function of the settings of the paging switches [*Appendix F1*] and the state of the User Address Compare Enable flag [*see below*].

Virtual addresses are supplied to the paging hardware via the address bus. An inadvertent failure occurs when the bus is not used for an access, but it accidentally contains the number set into the address switches. The data fetch switch also catches the attempt to retrieve a dispatch interrupt instruction or inadvertently a standard interrupt instruction, but the page failure sets the In-out Page Failure flag instead of resulting in a trap for an address failure.

Using this flag, the Monitor can return to a user instruction that caused an address failure and "set by it."

The data fetch switch was on and the requested access was for retrieval of an address word in an effective address calculation or read-only retrieval of an operand (other than in an XCT). This switch can also cause a failure inadvertently on the retrieval of a trap instruction or a PC word in an MUUO.

The write switch was on and the requested access was for writing, either write-only or read-modify-write, including writing by an LUUO (address 40). This switch also causes a failure on the first write in an MUUO if the address switches contain the effective address of the MUUO (even though that address is not used for the access), and can cause a failure inadvertently on the second write.

The Address Failure Inhibit flag, which can be set only by a JRSTF or MUUO, prevents an address failure during the next instruction – the completion of the next instruction automatically clears it. If an interrupt or trap intervenes, the flag has no effect and it is saved and cleared if the PC word is saved. If it is not saved, it affects the instruction following the interrupt or trap. Otherwise it affects the instruction following a return in which it is restored with the PC word.

Tests for hard page failures are actually made in the order given here.

22 Page refill failure – this is a hardware malfunction. The paging hardware did not find the virtual page listed in the page table, so it loaded paging information from the page map into the table but still could not find it.

20 Small user violation – a small user has attempted to reference a location outside of the limited small user address space.

21 Proprietary violation – an instruction in a public page has attempted to reference a concealed page or transfer control into a concealed page at an invalid entry point (one not containing a JRST 1,).

If the violation is not one of these, then bits 31-35 have the format shown above where *A*, *W* and *S* are simply the corresponding bits taken from the map half word for the page, and *T* indicates the type of reference in which the failure occurred – 0 for a read reference, 1 for a write or read-modify-write reference.

The type of reference implies nothing about the cause of failure – it indicates only the reason the failed reference was being made.

The page fail trap instruction is set by the Monitor to transfer control to kernel mode. After rectifying the situation, the Monitor returns to the interrupted instruction, which starts over again from the beginning. Even a two-part instruction that has been stopped by a failure in the second part is redone properly, provided the Monitor restores the First Part Done flag.

In a soft page failure, the mapping entry for the page is removed from the page table on the assumption that the Monitor will change it. When the instruction is restarted, the hardware must go to the page map to get a new entry for the page table.

Note that a failure does not necessarily imply that anything is "wrong". The virtual address space of even a small user is 32K words, which may well be more than is needed in a given run. Hence the Monitor may have only ten or twenty pages of the user program in core at any given time, and these would be the virtual pages indicated as accessible. When the user attempts to gain access to a page that is not there (a virtual page indicated in the page map as inaccessible), the Monitor would respond to the page failure by

bringing in the needed page from the drum or disk, either adding to the user space or swapping out a page the user no longer needs.

The same situation exists for writeability. When bringing in a user program, the Monitor would ordinarily indicate as writeable only the buffer area and other pages that will definitely be altered. Then in response to a write failure, the Monitor makes the page writeable and indicates to itself (perhaps by means of the software bit in the page map) that that page has in fact been altered. When the user is done, the Monitor need write only the altered pages back onto the drum.

Monitor Programming

The kernel mode program is responsible for the overall control of the system. It is the only program that has access to any of physical core unpaged and that has no instruction restrictions. The kernel program handles all in-out for the system and must set up the page maps, trap locations, interrupt locations and the like. The supervisor program labors under the same instruction restrictions as the user but has no way of bypassing them – they always apply. Supervisor mode is limited to the 144K paged part of the executive address space, although within that space it can read but not alter concealed pages (the kernel program supplies data tables of all kinds to the supervisor program, and the latter cannot affect them). The supervisor can give a JRSTF that clears Public provided it is also setting User; in other words the supervisor can return control to a concealed program but cannot enter kernel mode by manipulating the flags. The PC words supplied by MUUOs can manipulate the flags in any way, switching arbitrarily from one mode to another, but these are in the process table and assumed to be under control solely of kernel mode.

For accumulator, index register and fast memory references, the Monitor automatically uses fast memory block 0. For each user, the kernel mode program must assign a block. The usual procedure is to assign blocks 2 and 3 to individual user programs on a semipermanent basis for special applications and to assign block 1 to all other users. In this way the Monitor need not store blocks 2 and 3 when the special users are not running, and it need not store block 1 when it takes over control from an ordinary user temporarily. When switching from one user to another, the Monitor usually stores the first user's accumulators in his shadow area – this is locations 0–17 in user virtual page 0, an area not generally accessible to the user at all – and loads the new user's accumulators from his shadow area, where they were stored after the last time the new user ran.

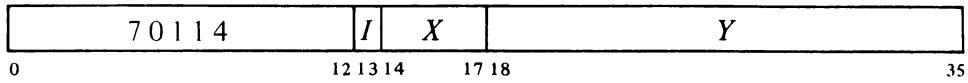
If the Monitor shared block 0 with any users, it would have to store the user accumulators even when taking control only temporarily.

Even while User is set, the interrupt instructions are not part of the user program and are thus subject only to executive restrictions. As interrupt instructions, JSR, JSP and PUSHJ automatically take the processor out of user mode to jump to an executive service routine. An MUUO can also be used.

The page failure and overflow trap instructions are executed in the user address space if caused by the user.

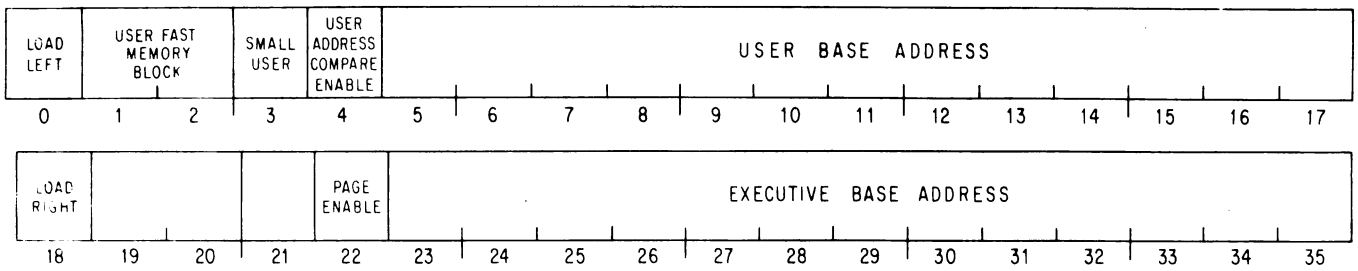
The paging hardware has one non-IO instruction and two condition IO instructions primarily for diagnostic purposes. Otherwise control over the system is exercised by data IO instructions. The device code for the paging hardware is 010, mnemonic PAG.

DATAO PAG, Data Out, Paging



Invalidating all data in the associative memory means setting the Word Empty bit in each location to indicate that the rest of the word is meaningless and should not be used.

Invalidate all data in the associative memory, and set up the paging hardware according to the contents of location *E* as shown.



Bits 0 and 18 are change bits. If bit 0 is 0, ignore the rest of the left half word. But if bit 0 is 1, load bits 5–17 into the user base register to select the user process table, select the fast memory block specified by bits 1 and 2 for the user, limit the address space to that of a small user if bit 3 is 1, and enable address comparison if bit 4 is 1.

The Address Compare Enable bit functions in conjunction with the console paging switches, as explained in Appendix F1.

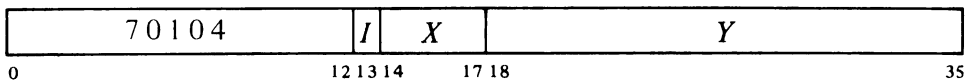
Similarly if bit 18 is 0, ignore the rest of the right half word. Otherwise load bits 23–35 into the executive base register to select the executive process table, and enable executive paging if bit 22 is 1. For normal operation of the system, bit 22 must be 1. A 0 in this bit disables overflow traps, and disables executive paging so there is no supervisor mode and no executive virtual addressing – in other words an executive mode program automatically runs in kernel mode with all access in the first 256K of physical memory unpaged.

An executive mode program that does not set bit 22 and avoids other special KI10 features will run on a KA10 as well. This is useful for hardware diagnostics and bootstrap loaders [*see readin mode*, §2.12].

NOTE

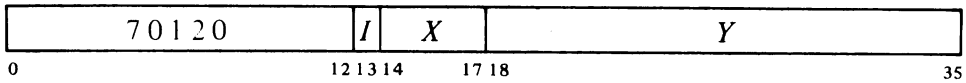
Neither turning on power nor pressing the reset switch invalidates the data in the associative memory. Therefore, after power has been off, the starting kernel mode program must do a DATAO PAG, to clear the associative memory of random data before entering executive or user paged address space.

DATAI PAG, Data In, Paging

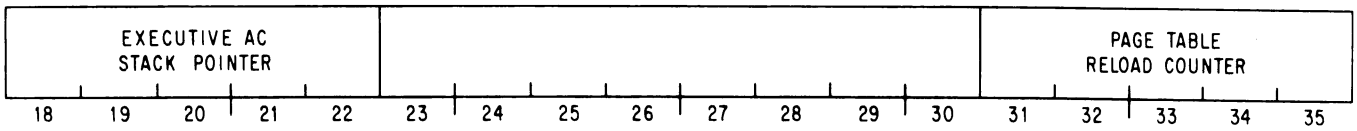


Read the status of the paging hardware into location *E*. The information read is the same as that supplied by a DATAO (bits 0 and 18 are 0).

CONO PAG, Conditions Out, Paging

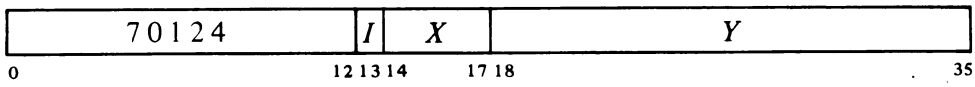


Load the executive stack pointer from bits 18–22 and the page table reload counter from bits 31–35 of the effective conditions *E* as shown.

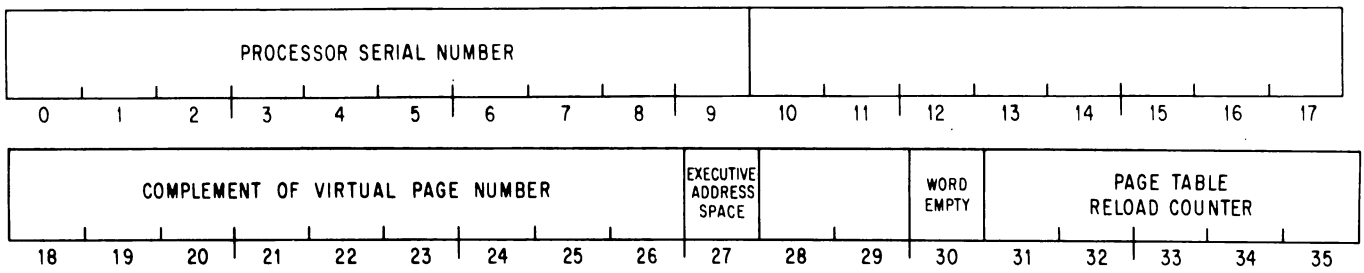


The executive stack pointer specifies a block of sixteen locations in the user process table by supplying the left five bits for a 9-bit address that references a location in the table; this function is used only for accessing stacked fast memory blocks in an instruction executed by an executive XCT [*see below*]. Loading the reload counter causes it to point to the specified location in the page table.

CONI PAG, Conditions In, Paging



Read the processor serial number, the page table reload counter, and the contents of the location in the virtual page table specified by the counter into the right half of location *E* as shown.

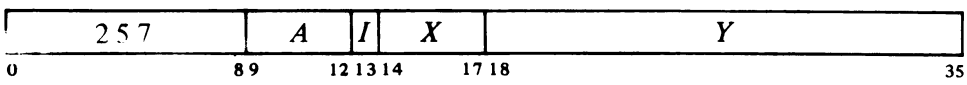


Note that bits 18–26 contain the complement of the virtual page number in the selected location. A 1 in bit 27 indicates the page is in the executive address space; a 1 in bit 30 means the information in bits 18–27 is invalid.

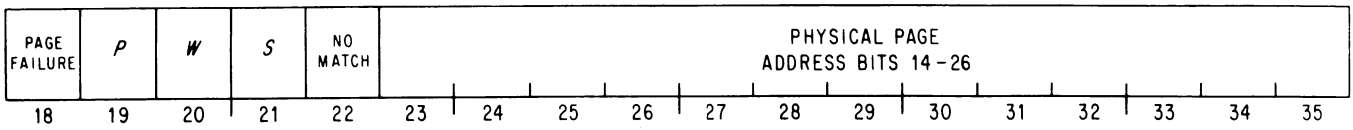
It is possible for the reload counter to change between the CONI and the CONO, so the CONI might read a different location than was selected by the CONO.

MAP Map an Address

Note that unlike all other instructions since §2.10, this is not an IO instruction.



Map the virtual effective address *E* and place the resulting map data in AC right in the same format as it is in the page map, *ie* bits *P*, *W* and *S* in bits 19–21 and the physical page number in bits 23–35. Clear AC left.



These three instructions can be used to inspect the contents of the associative memory. The CONO selects a location, the CONI reads the contents of the virtual-page part of that location, and an MAP that addresses the specified virtual page reads the contents of the physical-page part of that location.

This instruction cannot produce a page failure, but if a page failure would have resulted had an ordinary instruction in the same mode attempted to write in location *E*, place a 1 in AC bit 18. If no match can be made by the paging hardware, place a 1 in bit 22. This results in four possible situations as a function of the states of bits 18 and 22.

Bit 18	*Bit 22*	*Meaning*
0	0	AC right contains valid map data.
0	1	There is no page failure but also no match, so the instruction must have made an unmapped reference — perhaps to fast memory or to the unpaged area in kernel mode.
1	0	There is a page failure but the map data is correct as a match exists.
1	1	There is a page failure, and since there is no match, the failure must have resulted from the instruction referencing an inaccessible page or from some prior failure (such as a page refill malfunction). Hence AC right contains invalid information.

Executive XCT

Ordinarily an instruction in a user program is performed entirely in user address space and an instruction in the executive program is performed entirely in executive address space. In order to facilitate communication between Monitor and users, the XCT instruction allows the executive to execute instructions whose memory operand references can cross over the boundary between user and executive address spaces.

Read the next four paragraphs very carefully (reading them two or three times is highly recommended).

It is very important to note that the only difference between an instruction executed by an executive XCT and an instruction performed in normal circumstances is in the way the memory operand references are made. There is no difference in the XCT itself. Everything in the XCT is done in executive address space, and the instruction fetched by the XCT is fetched in executive space. Moreover, in the executed instruction all effective address calculation and accumulator references are in executive space. If the instruction makes no memory operand references, as in a jump, shift or immediate mode instruction, its execution differs in no way from the normal case. The only difference is in *memory operand references*.

Control over the special effects of the executed instructions is determined by the User In-out flag (whose implied meaning is confined to user mode) and bits 11 and 12 of the *A* portion of the XCT instruction word (in user mode *A* is ignored). If the *A* bits are both 0, the XCT acts as described in §2.9, and the executed instruction differs in no way from the normal case.

But if these bits are not both 0, then some memory operand references are made to *user* virtual address space, where the type of reference is determined by the *A* bits and the type of memory is selected by User In-out. With this flag set, the *A* bits affect both core memory and fast memory references, whereas with User In-out clear, the *A* bits affect only fast memory references. For the memory operand references selected by User In-out, the effect of 1s in bits 11 and 12 is as follows: a 1 in bit 12 causes the executed instruction to perform all selected read and read-modify-write memory operand references to be performed in user virtual address space; a 1 in bit 11 causes all selected memory operand write references to be performed in user space; and 1s in both bits cause all types of selected memory operand references in the executed instruction to be performed in user space.

The meaning of user space is obvious in terms of core memory references, but not so for fast memory. When User In-out is set, the user space for fast memory references depends on which fast memory block is currently selected for the user. If block 0 is selected, fast memory operand references of the types specified by bits 11 and 12 are made to the user shadow area. If some other block is selected, the specified fast memory references are made to the selected block.

If User In-out is clear, all core memory references are in executive address space. Fast memory references of the types specified by bits 11 and 12 are made to the user process table, in particular to that set of sixteen locations specified by the executive stack pointer. The pointer is given by a CONO PAG,.

User Space Fast Memory References

	User Fast Memory Block Selected	
User In-out	*Zero*	*Nonzero*
1	Shadow area	Selected block
0	AC stack	AC stack

There is another flag that plays a role in the execution of instructions by an executive XCT. This is Disable Bypass, bit 0 of the PC word. When Disable Bypass is clear, a bypass in the logic allows an executed instruction to access the concealed user area from supervisor mode. With the flag set, an attempt to do this results in a page failure. Generally the new MUUO PC word would set this flag when the Monitor is being called from public mode, so the concealed area can be accessed only when such access is requested by the concealed program.

Individual Instruction Effects. The effects of execution by an executive XCT on different types of instructions is as follows.

◆ Instructions without memory operand references are not affected. This includes shifts, jumps, immediate mode instructions, CONSO, CONO, and even an XCT. In fact not only is an executive XCT not affected when executed by an executive XCT, but the first destroys any effect the second would otherwise have on a third instruction (in other words, a pair of executive XCTs is equivalent to a pair of ordinary XCTs).

♦ Instructions that refer to one memory location for reading only or reading and writing are controlled by the read bit (MOVE, MOVES, ADDM, AOS). The read bit controls writing when the write is done to the same location as the read, whether the memory references are done as a single cycle including both read and write or as separate read and write cycles.
♦ Instructions that refer to one memory location for writing only are controlled by the write bit (MOVEM, MAP, HRLZM).
♦ Instructions that refer to two different memory locations are controlled by the read bit in the read part of the instruction and by the write bit in the write part (BLT, PUSH).
♦ BLKI and BLKO are controlled by the write bit and the read bit respectively. The pointer reference is done in the same address space as the data transfer.
♦ In byte instructions all pointer calculations are done in executive address space. The read and write bits affect only the second part, *ie* the load or deposit.

Philosophy. The purpose of the executive XCT is to facilitate the handling of user requirements by the Monitor, but the selection made by User In-out of the references affected by the read and write bits is to allow the Monitor to make recursive calls to itself, *ie* to perform MUUOs in the process of carrying out an MUUO given by the user. Specifically the state of User In-out differentiates between the Monitor response directly to the user MUUO and its response to its own MUUOs.

The new PC word of an MUUO from the user would set User In-out so that core memory references can be made across the user-executive boundary, and fast memory references can be made to the user AC block. The point in choosing between the shadow area and the selected block if not block 0 is to reference the information that was held in the user AC block before the Monitor took over. If the user shared block 0 with other users and the Monitor, the Monitor will have saved his ACs in the shadow area of his address space. The other AC blocks are not disturbed when the Monitor takes over temporarily, so the Monitor need not save them and they will still hold the user information.

If in the course of carrying out a user MUUO, the Monitor should itself give an MUUO, the new PC word would clear User In-out. Thus at this level all core memory references are in the executive address space and fast memory references are to an AC block in the user process table as specified by the executive stack pointer. MUUO calls by the Monitor to itself can be nested to a number of levels, but in all cases User In-out is left clear. The particular AC block used at any level is specified by the stack pointer. Hence the AC stack in the user process table is effectively a pushdown list kept by the stack pointer; at each level the program must change the pointer to specify the appropriate block. Each user process table would contain the blocks needed for carrying out MUUOs for that user.

This makes a different set of sixteen words available at each level using the same addresses.

EXAMPLE. Suppose that the Monitor has been called by an MUUO from the user (hence User In-out is set) and wishes to save the user's ACs in the shadow area. Assume that every user runs with AC block 1, 2 or 3, and that the Monitor always sets up executive virtual page 342 to point to the same

physical page as user page 0. Using accumulator T in block 0, the Monitor saves the user ACs by giving these two instructions,

```
MOVEI   T,342000        ;Initialize pointer: from 0 to 342000
XCT     1,[BLT  T,342017]
```

and restores them with these two.

```
MOVSI   T,342000        ;From 342000 to 0
XCT     2,[BLT  T,17]
```

2.17 KA10 PROGRAM AND MEMORY MANAGEMENT

The KA10 has only user and executive modes and uses protection and relocation hardware.

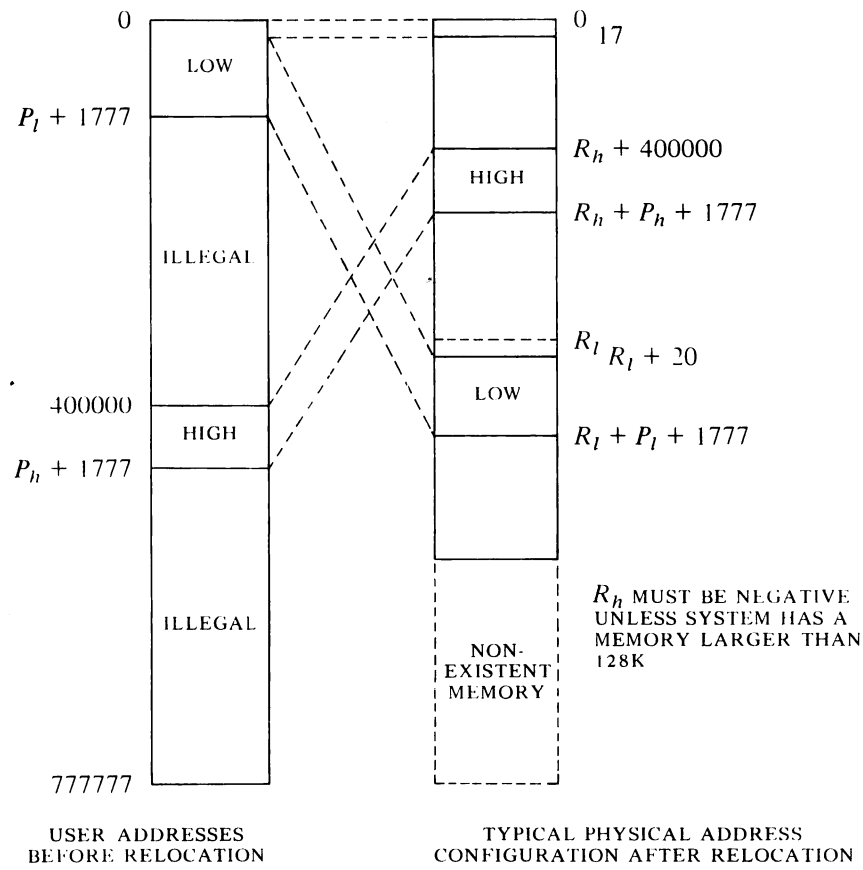
Every user is assigned a core area and the rest of core is protected from him – he cannot gain access to the protected area for either storage or retrieval of information. The assigned area is divided into two parts. The low part is unique to a given user and can be used for any purpose. The high part may be for a single user, or it may be shared by several users. The Monitor can write-protect the high part so that the user cannot alter its contents, *ie* he cannot write anything in it. The Monitor would do this when the high part is to be a pure procedure to be used reentrantly by several users. One high pure segment may be used with any number of low impure segments. The user can request that the Monitor write-protect the high part of a single program, *eg* in order to debug a reentrant program. All users write programs beginning at address 0 for the low part, and beginning usually at 400000 for the high part. The programmed addresses are retained in the object program but are relocated by the hardware to the physical area assigned to the user as each access is made while the program is running.

The size and position of the user area are defined by specifying protection and relocation addresses for the low and high blocks. The protection address determines the maximum address the user can give; any address larger than the maximum is illegal. The relocation address is the address, as seen by the Monitor and the hardware, of the first location in the block. The Monitor defines these addresses by loading four 8-bit registers, each of which corresponds to the left eight bits (18–25) of an address whose right ten bits are all 0.

To determine whether an address is legal its left eight bits are compared with the appropriate protection register, so the maximum user address consists of the register contents in its left eight bits, 1777 in its right ten bits (*ie* it is equal to the protection address plus 1777). Since the set of all addresses begins at zero, a block is always an integral multiple of 1024_{10} (2000_8) locations. Relocation is accomplished simply by adding the contents of the appropriate relocation register to the user address, so the first address in a block is a multiple of 2000. The relative user and relocated address configurations are therefore as illustrated here, where P_l, R_l, P_h and R_h are respectively the protection and relocation addresses for the low and high

Note that the relocated low part is actually in two sections with the larger beginning at $R_l + 20$. This is because addresses 0–17 are not relocated, all users having access to the accumulators. The Monitor uses the first sixteen locations in the low user block to store the user's accumulators when his program is not running.

Some systems have only the low pair of protection and relocation registers. In this case the user program is always nonreentrant and the assigned area comprises only the low part.



parts as derived from the 8-bit registers loaded by the Monitor. If the low part is larger than 128K locations, *ie* more than half the maximum memory capacity ($P_l \geqslant 400000$), the high part starts at the first location after the low part (at location $P_l + 2000$). The high part is limited to 128K. If the Monitor defines two parts but does not write-protect the high part, the user has a two-part nonreentrant program.

If the user attempts to access a location outside of his assigned area, or if the high part is write-protected and he attempts to alter its contents, the current instruction terminates immediately, the Memory Protection flag is set (status bit 22 read by CONI APR,), and an interrupt is requested on the channel assigned to the processor [§2.14].

Addressing Summary. Let A_u be the address supplied by the user, and let A_p be the physical core address generated from it by the relocation hardware.

If a relocated address is in the range 0–17, the reference is to core rather than fast memory.

If $A_u \leqslant 17$, then $A_p = A_u$ (fast memory, no relocation).

If $20 \leqslant A_u \leqslant P_l + 1777$, then $A_p = (A_u + R_l) \bmod 2^{18}$.

If the greater of $\left\{\begin{matrix}400000\\P_l + 2000\end{matrix}\right\} \leqslant A_u \leqslant P_h + 1777$,

then $A_p = (A_u + R_h) \bmod 2^{18}$.

Any other value of A_u is illegal. These are $A_u > P_l + 1777$ if either $A_u < 400000$ or $A_u > P_h + 1777$.

User Programming. The user must observe the following rules when programming on a time shared basis. [*Refer to the Monitor manual for further information including use of the Monitor for input-output.*]

The user can actually write any size program: the Monitor will assign enough core for his needs. Basically the user must write a sensible program; if he uses absolute addresses scattered all over memory his program cannot be run on a time shared basis with others.

- Use addresses only within the assigned blocks for all purposes – retrieval of instructions, retrieval of addresses, storage or retrieval of operands. The low part contains locations with addresses from 0 to the maximum; the high part contains from the greater of 400000 or $P_l + 2000$ to the maximum. Either part can address the other.
- If the high part is write-protected, do not attempt to store anything in it. In particular do not execute a JSR or JSA into the high part.
- Use instruction codes 000 and 040–127 only in the manner prescribed in the Monitor manual.
- Unless User In-out is set do not give any IO instruction, HALT (JRST 4,) or JEN (JRST 12, (specifically JRST 10,)). The program can determine if User In-out is set by examining bit 6 of the PC word stored by JSR, JSP or PUSHJ.

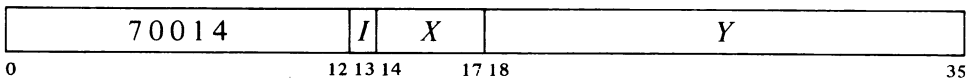
These instructions are illegal unless User In-out is set.

The user can give a JRSTF (JRST 2,) but a 0 in bit 5 of the PC word does not clear User (a program cannot leave user mode this way); and a 1 in bit 6 does not set User In-out, so the user cannot void any of the restrictions himself. Note that a 0 in bit 6 will clear User In-out, so a user can discard his own special privileges.

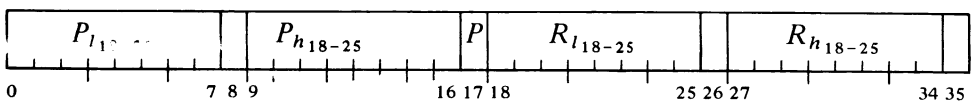
LUUOs (001–037) function normally and are relocated to addresses 40 and 41 in the low block [§2.10].

Monitor Programming. The Monitor must assign the core area for each user program, set up trap and interrupt locations, specify whether the user can give IO instructions, transfer control to the user program, and respond appropriately when an interrupt occurs or an instruction is executed in unrelocated 41 or 61. Core assignment is made by this instruction.

DATAO APR, Data Out, Arithmetic Processor



Load the protection and relocation registers from the contents of location E as shown, where P_l, P_h, R_l and R_h are the protection and relocation



addresses defined above. If write-protect bit P (bit 17) is 1, do not allow the user to write in the high part of his area.

For a two part nonreentrant program, set $P = 0$. For a one-part nonreentrant program, make $P_h \leq P_l$. If the hardware has only one set of protection and relocation registers, the user area is defined by P_l and R_l, the rest of the word is ignored.

Giving a JRSTF with a 1 in bit 6 of the PC word allows the user to handle his own input-output. The Monitor can also transfer control to the user with this instruction by programming a 1 in bit 5 of the PC word, or it may jump to the user program with a JRST 1, which automatically sets User. The set state of this flag implements the user restrictions.

While User is set, certain instructions are not part of the user program and are therefore completely unrestricted, namely those executed in the interrupt locations (which are not relocated) and in unrelocated trap locations 41 and 61. Illegal instructions and UUO codes 000 and 040–077 are trapped in unrelocated 40; codes 100–127 are trapped in unrelocated 60. BLKI and BLKO can be used in the even interrupt locations, and if there is no overflow, the processor returns to the interrupted user program. JSR should ordinarily be used in the remaining even interrupt locations, in odd interrupt locations following block IO instructions, and in 41 and 61. The JSR clears User and should jump to the Monitor. JSP, PUSHJ, JSA and JRST are acceptable in that they clear User, but the first two require an accumulator (all accumulators should be available to the user) and the latter two do not save the flags.

The trap locations are 140-141 and 160-161 in a second KA10 processor.

After taking appropriate action, the Monitor can return to the user program with a JRSTF or JEN that restores the flags including User and User In-out.

2.18 REAL TIME CLOCK DK10

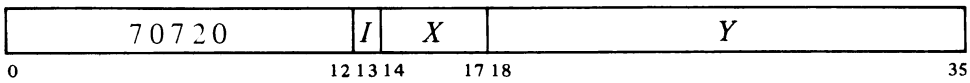
The clock referred to throughout this section is the DK10 real time clock and should not be confused with the line frequency clock whose flag is one of the processor conditions [§2.14].

This processor option can be used to signal the end of a specified real time interval or to measure the real time taken by an event. With appropriate software the DK10 can easily be used to keep the time of day. The basic element in the clock is an 18-bit binary counter that is incremented repeatedly by a clock source; a 100 kHz ± .01% crystal-controlled source is available internally, or a source of any frequency up to 400 kHz can be provided externally. Operation is synchronized so that the program can read the counter at any time without missing a count. Associated with the counter is an 18-bit interval register, which can be loaded by the program. Each time the count reaches the number held in the register, the clock requests an interrupt while the counter clears and begins a new count. With the internal clock source, whose period is 10 μs, the total count is about 2.6 seconds.

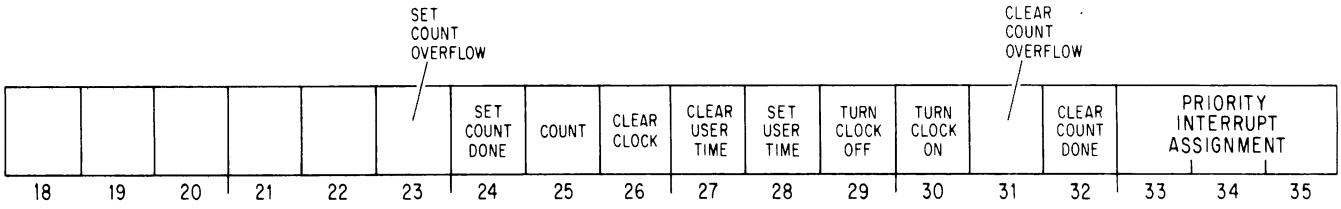
The program turns the clock on and off by enabling and disabling the counter. The clock has two modes of operation: with the User Time flag clear, the counter operates continuously; with User Time set, the counter stops while the processor is handling interrupts. Hence in the latter mode the clock discounts interrupt time and can be used to time user programs. In a system that contains two clocks, one can be used by the Monitor to time user programs while the other is used to keep the time of day.

Instructions. The clock device code is 070, mnemonic CLK. A second clock would have device code 074.

CONO CLK, Conditions Out, Clock



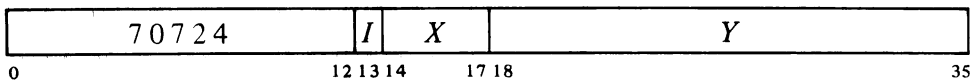
Assign the interrupt channel specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 23–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



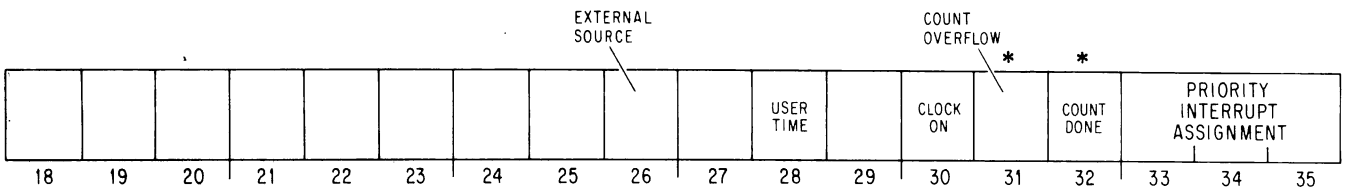
A 1 in bit 26 clears the clock counter and the Count Done, Count Overflow and User Time flags, turns off the clock, and dismisses the PI assignment (assigns zero). The effect of giving conflicting conditions is indeterminate.

A 1 in bit 25 increments the counter provided the clock is off (this is for maintenance only).

CONI CLK, Conditions In, Clock



Read the contents of the interval register into the left half of location *E* and read the status of the clock into bits 26–35 as shown.



Interrupts are requested on the assigned channel by the setting of Count Overflow and Count Done.

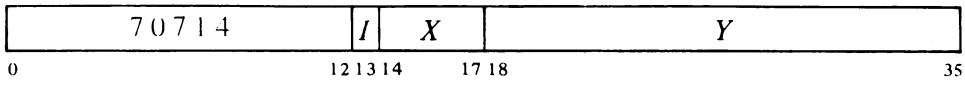
26 The counter is connected to an external source (0 indicates the internal source is connected).

28 The counter cannot be incremented while an interrupt is being held or a request has been accepted and the channel is waiting for an interrupt to start.

*These bits cause interrupts.

Note that to time a user properly, the Monitor must also compensate for any noninterrupt time taken from the user.

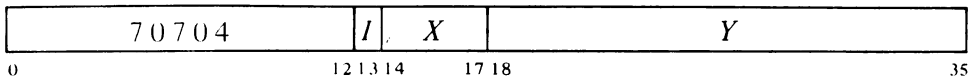
DATAO CLK, Data Out, Clock



Load the contents of the right half of location *E* into the interval register.

The comparison of the counter against the interval register that follows every count is inhibited while this instruction is loading the register.

DATAI CLK, Data In, Clock



Read the current contents of the clock counter into the right half of location *E*.

The counter is always stable while being read, and any count held back is picked up immediately afterward.

Initially the program should give a CONO CLK,1000 to clear the clock, and then give a DATAO to select the interval and a CONO to turn on the clock, select the mode, and assign the interrupt channel. When the count reaches the specified interval, Count Done sets, requesting an interrupt on the assigned channel. At the same time, the counter clears and a new count begins with the next pulse. The program should respond with a CONO to clear Count Done.

Following turnon the first count may occur at any time up to the full period of the source.

Remember that although a CONO need not affect the mode or the clock state, every CONO must renew the PI assignment.

The interval can be changed at any time simply by giving a DATAO. However, if the program does not clear the counter at the same time, then it should make sure that the count has not yet reached the value of the new interval. If the count is already beyond that point, the counter will continue until it overflows. When the counter overflows, either because the count started too high, the program specified the maximum count (2^{18} is selected by loading zero), or there is a malfunction of some sort, Count Overflow sets, requesting an interrupt, and a new count begins.

To use the clock to time some operation, turn it on with the counter at zero. For a counter reading of *C*, the elapsed time is

$$T(C + nI)$$

where *T* is the period of the source, *n* is the number of clock interrupts since the clock was started, and *I* is the interval selected by the program. To cause the clock to request an interrupt after $T \times n$ μs, where $n \leq 2^{18}$ and *T* is the period of the source in microseconds, load the interval register with *n* expressed in binary. There is an average indeterminacy of half a count every time the counter starts and stops. Therefore, when the clock is keeping user time, there is an average indeterminacy of one count for every *group* of overlapping interrupts and requests (not for every interrupt, as the counter is inhibited while there is any request or interrupt being held).

For keeping the time of day, the program can use a memory location to maintain a count of the clock interrupts. The location should be cleared

at midnight, and the time can be determined by combining its contents with the current contents of the clock counter. If the location itself is to be used as a low resolution clock kept in hours, minutes and seconds, it is better to use a more convenient interval than the full count. Using the internal source, an interval of 2½ seconds, which is octal 750220, is the most straightforward interval with the fewest interrupts. To interrupt every second the interval would be 303240.

Note that an error of .01% amounts to 8.64 seconds in 24 hours.

Appendices

APPENDIX A

INSTRUCTIONS AND MNEMONICS

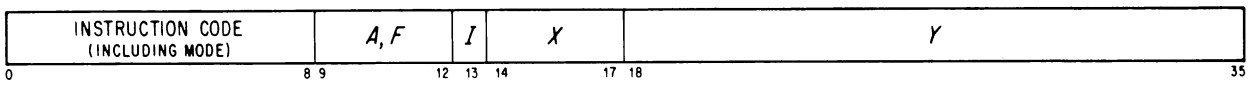
The drawing on the next two pages shows the formats of the various types of words used by the processor (instructions, pointers, operands, etc). The illustration on page A-4 shows the derivation of the instruction mnemonics. Next are two tables that list all instruction mnemonics and their octal codes both numerically and alphabetically. When two mnemonics are given for the same octal code, the first is the preferred form, but the assembler does recognize the second. For completeness, the tables include the MUUOs (indicated by an asterisk) that are recognized by MACRO for communication with the DECsystem-10 Time Sharing Monitor. A double dagger (‡) indicates a KI10 instruction code that is unassigned in the KA10.

Note that 247 is unassigned in the KI10, but 247 and 257 execute as no-ops in the KA10.

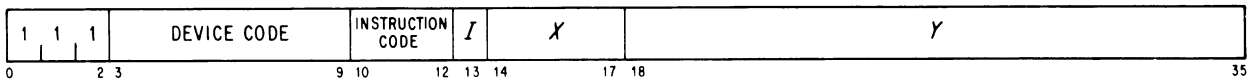
In-out device codes are included only in the alphabetic listing and are indicated by a dagger (†). Following the tables is a chart that lists the devices with their mnemonic and octal codes and DEC type numbers for both PDP-10 and PDP-6. A device mnemonic ending in the numeral 2 is the recommended form for the second of a given device, but such codes are not recognized by MACRO – they must be defined by the user.

Beginning on page A-13 is a list of all instructions showing their actions in symbolic form. On page A-23 is a table of the positive and negative powers of 2.

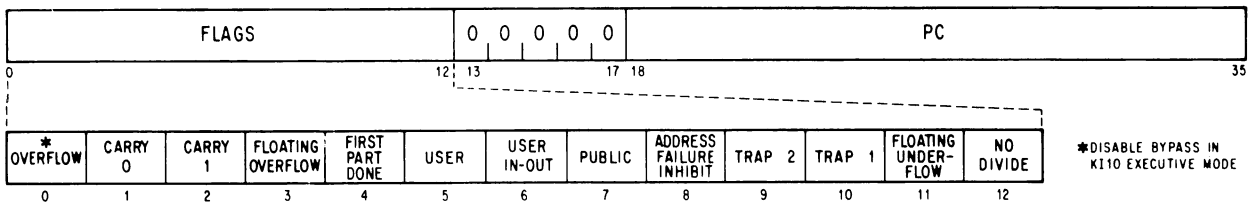
BASIC INSTRUCTIONS



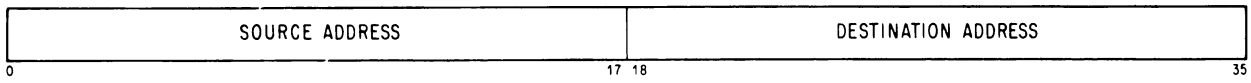
IN-OUT INSTRUCTIONS



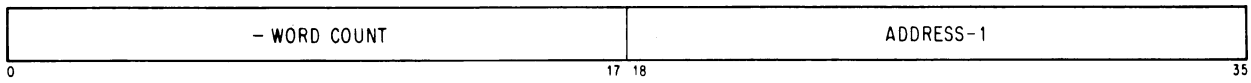
PC WORD



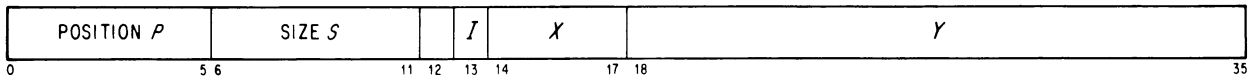
BLT POINTER {XWD}



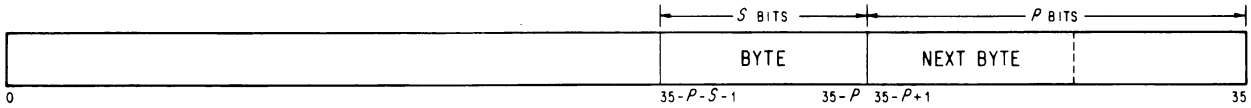
BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



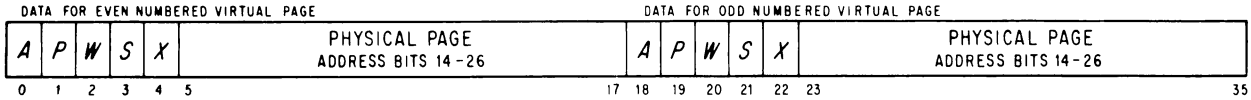
BYTE POINTER



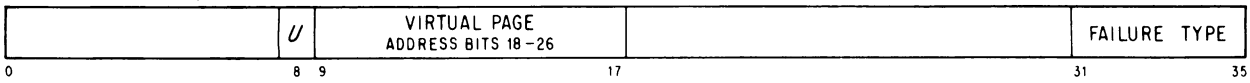
BYTE STORAGE



PAGE MAP WORD

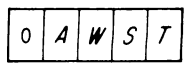


PAGE FAIL WORD

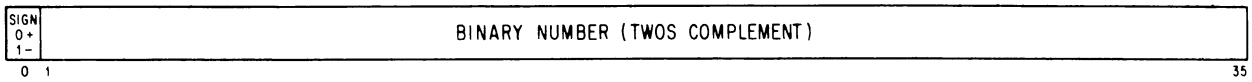


20 SMALL USER VIOLATION
21 PROPRIETARY VIOLATION
22 PAGE REFILL FAILURE
23 ADDRESS FAILURE

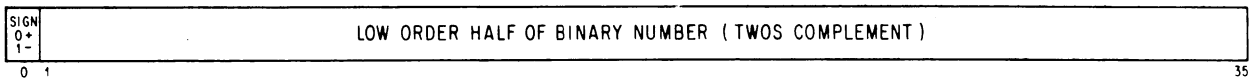
IF BIT 31 IS 0, BITS 31-35 HAVE THIS FORMAT



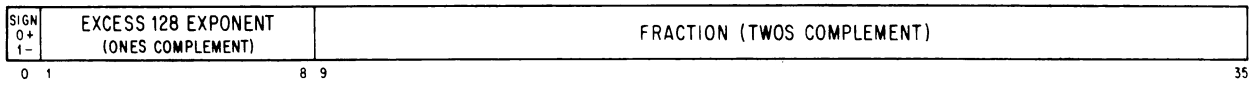
FIXED POINT OPERANDS (SINGLE PRECISION OR HIGH ORDER WORD)



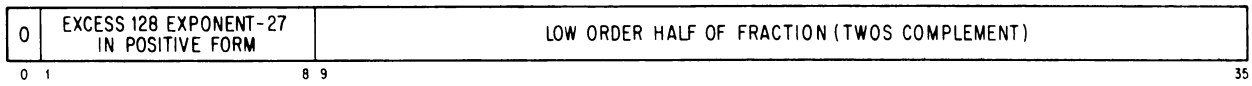
LOW ORDER WORD IN DOUBLE LENGTH FIXED POINT OPERANDS



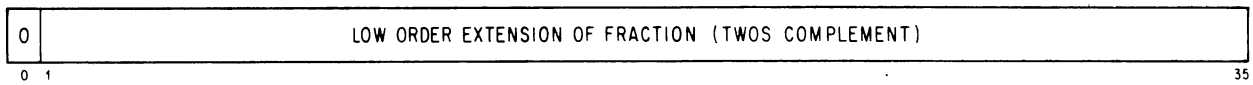
FLOATING POINT OPERANDS (SINGLE PRECISION OR HIGH ORDER WORD)

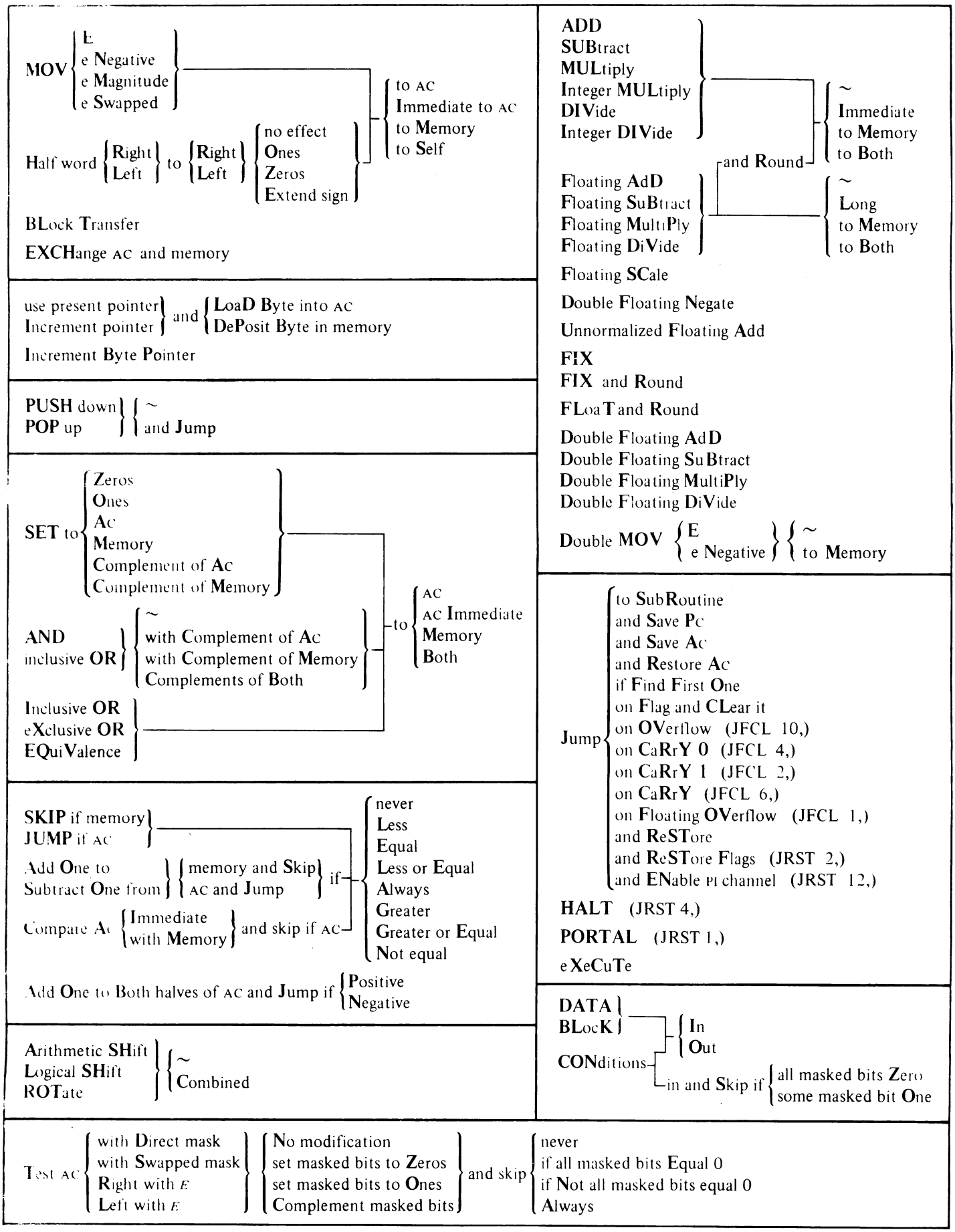


LOW ORDER WORD IN SOFTWARE DOUBLE LENGTH FLOATING POINT OPERANDS



LOW ORDER WORD IN HARDWARE DOUBLE LENGTH FLOATING POINT OPERANDS




MOV
E
e Negative
e Magnitude
e Swapped
to AC
Immediate to AC
to Memory
to Self
Half word
Right
Left
to
Right
Left
no effect
Ones
Zeros
Extend sign
BLock Transfer
EXCHange AC and memory
use present pointer
Increment pointer
and
LoaD Byte into AC
DePosit Byte in memory
Increment Byte Pointer
PUSH down
POP up
~
and Jump
SET to
Zeros
Ones
Ac
Memory
Complement of Ac
Complement of Memory
AND
inclusive OR
~
with Complement of Ac
with Complement of Memory
Complements of Both
to
AC
AC Immediate
Memory
Both
Inclusive OR
eXclusive OR
EQuiValence
SKIP if memory
JUMP if AC
Add One to
Subtract One from
memory and Skip
AC and Jump
if
never
Less
Equal
Less or Equal
Always
Greater
Greater or Equal
Not equal
Compare Ac
Immediate
with Memory
and skip if AC
Add One to Both halves of AC and Jump if
Positive
Negative
Arithmetic SHift
Logical SHift
ROTate
~
Combined
ADD
SUBtract
MULtiply
Integer MULtiply
DIVide
Integer DIVide
and Round
~
Immediate
to Memory
to Both
Floating AdD
Floating SuBtract
Floating MultiPly
Floating DiVide
and Round
~
Long
to Memory
to Both
Floating SCale
Double Floating Negate
Unnormalized Floating Add
FIX
FIX and Round
FLoaT and Round
Double Floating AdD
Double Floating SuBtract
Double Floating MultiPly
Double Floating DiVide
Double MOV
E
e Negative
~
to Memory
Jump
to SubRoutine
and Save Pc
and Save Ac
and Restore Ac
if Find First One
on Flag and CLear it
on OVerflow (JFCL 10,)
on CaRrY 0 (JFCL 4,)
on CaRrY 1 (JFCL 2,)
on CaRrY (JFCL 6,)
on Floating OVerflow (JFCL 1,)
and ReSTore
and ReSTore Flags (JRST 2,)
and ENable PI channel (JRST 12,)
HALT (JRST 4,)
PORTAL (JRST 1,)
eXeCuTe
DATA
BLocK
In
Out
CONditions
in and Skip if
all masked bits Zero
some masked bit One
Test AC
with Direct mask
with Swapped mask
Right with E
Left with E
No modification
set masked bits to Zeros
set masked bits to Ones
Complement masked bits
and skip
never
if all masked bits Equal 0
if Not all masked bits equal 0
Always

INSTRUCTION MNEMONICS

NUMERIC LISTING

000	ILLEGAL	106		162	FMPM
001		107		163	FMPB
:	LUUO'S	110	‡DFAD	164	FMPR
:		111	‡DFSB	165	FMPRI
037		112	‡DFMP	166	FMPRM
040	*CALL	113	‡DFDV	167	FMPRB
041	*INIT	114		170	FDV
042		115		171	FDVL
043	RESERVED FOR SPECIAL MONITORS	116		172	FDVM
044		117		173	FDVB
045		120	‡DMOVE	174	FDVR
046		121	‡DMOVN	175	FDVRI
047	*CALLI	122	‡FIX	176	FDVRM
050	*OPEN	123		177	FDVRB
051	*TTCALL	124	‡DMOVEM	200	MOVE
052		125	‡DMOVNM	201	MOVEI
053	RESERVED FOR DEC	126	‡FIXR	202	MOVEM
054		127	‡FLTR	203	MOVES
055	*RENAME	130	UFA	204	MOVS
056	*IN	131	DFN	205	MOVSI
057	*OUT	132	FSC	206	MOVSM
060	*SETSTS	133	IBP	207	MOVSS
061	*STATO	134	ILDB	210	MOVN
062	*STATUS	135	LDB	211	MOVNI
062	*GETSTS	136	IDPB	212	MOVNM
063	*STATZ	137	DPB	213	MOVNS
064	*INBUF	140	FAD	214	MOVM
065	*OUTBUF	141	FADL	215	MOVMI
066	*INPUT	142	FADM	216	MOVMM
067	*OUTPUT	143	FADB	217	MOVMS
070	*CLOSE	144	FADR	220	IMUL
071	*RELEAS	145	FADRI	221	IMULI
072	*MTAPE	146	FADRM	222	IMULM
073	*UGETF	147	FADRB	223	IMULB
074	*USETI	150	FSB	224	MUL
075	*USETO	151	FSBL	225	MULI
076	*LOOKUP	152	FSBM	226	MULM
077	*ENTER	153	FSBB	227	MULB
100	*UJEN	154	FSBR	230	IDIV
101		155	FSBRI	231	IDIVI
102		156	FSBRM	232	IDIVM
103		157	FSBRB	233	IDIVB
104		160	FMP	234	DIV
105		161	FMPL	235	DIVI

236	DIVM	306	CAIN	367	SOJG
237	DIVB	307	CAIG	370	SOS
240	ASH	310	CAM	371	SOSL
241	ROT	311	CAML	372	SOSE
242	LSH	312	CAME	373	SOSLE
243	JFFO	313	CAMLE	374	SOSA
244	ASHC	314	CAMA	375	SOSGE
245	ROTC	315	CAMGE	376	SOSN
246	LSHC	316	CAMN	377	SOSG
247		317	CAMG	400	SETZ
250	EXCH	320	JUMP	400	CLEAR
251	BLT	321	JUMPL	401	SETZI
252	AOBJP	322	JUMPE	401	CLEARI
253	AOBJN	323	JUMPLE	402	SETZM
254	JRST	324	JUMPA	402	CLEARM
25404	PORTAL	325	JUMPGE	403	SETZB
25410	JRSTF	326	JUMPN	403	CLEARB
25420	HALT	327	JUMPG	404	AND
25450	JEN	330	SKIP	405	ANDI
255	JFCL	331	SKIPL	406	ANDM
25504	JFOV	332	SKIPE	407	ANDB
25510	JCRY1	333	SKIPLE	410	ANDCA
25520	JCRY0	334	SKIPA	411	ANDCAI
25530	JCRY	335	SKIPGE	412	ANDCAM
25540	JOV	336	SKIPN	413	ANDCAB
256	XCT	337	SKIPG	414	SETM
257	‡MAP	340	AOJ	415	SETMI
260	PUSHJ	341	AOJL	416	SETMM
261	PUSH	342	AOJE	417	SETMB
262	POP	343	AOJLE	420	ANDCM
263	POPJ	344	AOJA	421	ANDCMI
264	JSR	345	AOJGE	422	ANDCMM
265	JSP	346	AOJN	423	ANDCMB
266	JSA	347	AOJG	424	SETA
267	JRA	350	AOS	425	SETAI
270	ADD	351	AOSL	426	SETAM
271	ADDI	352	AOSE	427	SETAB
272	ADDM	353	AOSLE	430	XOR
273	ADDB	354	AOSA	431	XORI
274	SUB	355	AOSGE	432	XORM
275	SUBI	356	AOSN	433	XORB
276	SUBM	357	AOSG	434	IOR
277	SUBB	360	SOJ	434	OR
300	CAI	361	SOJL	435	IORI
301	CAIL	362	SOJE	435	ORI
302	CAIE	363	SOJLE	436	IORM
303	CAILE	364	SOJA	436	ORM
304	CAIA	365	SOJGE	437	IORB
305	CAIGE	366	SOJN	437	ORB

440	ANDCB	521	HLLOI	602	TRNE
441	ANDCBI	522	HLLOM	603	TLNE
442	ANDCBM	523	HLLOS	604	TRNA
443	ANDCBB	524	HRLO	605	TLNA
444	EQV	525	HRLOI	606	TRNN
445	EQVI	526	HRLOM	607	TLNN
446	EQVM	527	HRLOS	610	TDN
447	EQVB	530	HLLE	611	TSN
450	SETCA	531	HLLEI	612	TDNE
451	SETCAI	532	HLLEM	613	TSNE
452	SETCAM	533	HLLES	614	TDNA
453	SETCAB	534	HRLE	615	TSNA
454	ORCA	535	HRLEI	616	TDNN
455	ORCAI	536	HRLEM	617	TSNN
456	ORCAM	537	HRLES	620	TRZ
457	ORCAB	540	HRR	621	TLZ
460	SETCM	541	HRRI	622	TRZE
461	SETCMI	542	HRRM	623	TLZE
462	SETCMM	543	HRRS	624	TRZA
463	SETCMB	544	HLR	625	TLZA
464	ORCM	545	HLRI	626	TRZN
465	ORCMI	546	HLRM	627	TLZN
466	ORCMM	547	HLRS	630	TDZ
467	ORCMB	550	HRRZ	631	TSZ
470	ORCB	551	HRRZI	632	TDZE
471	ORCBI	552	HRRZM	633	TSZE
472	ORCBM	553	HRRZS	634	TDZA
473	ORCBB	554	HLRZ	635	TSZA
474	SETO	555	HLRZI	636	TDZN
475	SETOI	556	HLRZM	637	TSZN
476	SETOM	557	HLRZS	640	TRC
477	SETOB	560	HRRO	641	TLC
500	HLL	561	HRROI	642	TRCE
501	HLLI	562	HRROM	643	TLCE
502	HLLM	563	HRROS	644	TRCA
503	HLLS	564	HLRO	645	TLCA
504	HRL	565	HLROI	646	TRCN
505	HRLI	566	HLROM	647	TLCN
506	HRLM	567	HLROS	650	TDC
507	HRLS	570	HRRE	651	TSC
510	HLLZ	571	HRREI	652	TDCE
511	HLLZI	572	HRREM	653	TSCE
512	HLLZM	573	HRRES	654	TDCA
513	HLLZS	574	HLRE	655	TSCA
514	HRLZ	575	HLREI	656	TDCN
515	HRLZI	576	HLREM	657	TSCN
516	HRLZM	577	HLRES	660	TRO
517	HRLZS	600	TRN	661	TLO
520	HLLO	601	TLN	662	TROE

663	TLOE	673	TSOE	70010	BLKO
664	TROA	674	TDOA	70014	DATAO
665	TLOA	675	TSOA	70020	CONO
666	TRON	676	TDON	70024	CONI
667	TLON	677	TSON	70030	CONSZ
670	TDO	70000	BLKI	70034	CONSO
671	TSO	70004	DATAI		
672	TDOE	70004	RSW		

INSTRUCTION MNEMONICS

ALPHABETIC LISTING

†ADC	024	AOSA	354	†CDP	110
ADD	270	AOSE	352	†CDR	114
ADDB	273	AOSG	357	CLEAR	400
ADDI	271	AOSGE	355	CLEARB	403
ADDM	272	AOSL	351	CLEARI	401
AND	404	AOSLE	353	CLEARM	402
ANDB	407	AOSN	356	†CLK	070
ANDCA	410	†APR	000	*CLOSE	070
ANDCAB	413	ASH	240	CONI	70024
ANDCAI	411	ASHC	244	CONO	70020
ANDCAM	412	BLKI	70000	CONSO	70034
ANDCB	440	BLKO	70010	CONSZ	70030
ANDCBB	443	BLT	251	†CPA	000
ANDCBI	441	CAI	300	†CR	150
ANDCBM	442	CAIA	304	DATAI	70004
ANDCM	420	CAIE	302	DATAO	70014
ANDCMB	423	CAIG	307	†DC	200
ANDCMI	421	CAIGE	305	†DCSA	300
ANDCMM	422	CAIL	301	†DCSB	304
ANDI	405	CAILE	303	‡DFAD	110
ANDM	406	CAIN	306	‡DFDV	113
AOBJN	253	*CALL	040	‡DFMP	112
AOBJP	252	*CALLI	047	DFN	131
AOJ	340	CAM	310	‡DFSB	111
AOJA	344	CAMA	314	†DIS	130
AOJE	342	CAME	312	DIV	234
AOJG	347	CAMG	317	DIVB	237
AOJGE	345	CAMGE	315	DIVI	235
AOJL	341	CAML	311	DIVM	236
AOJLE	343	CAMLE	313	†DLB	060
AOJN	346	CAMN	316	†DLC	064
AOS	350	†CCI	014	†DLS	240

JRST	254
JRSTF	25410
JSA	266
JSP	265
JSR	264
JUMP	320
JUMPA	324
JUMPE	322
JUMPG	327
JUMPGE	325
JUMPL	321
JUMPLE	323
JUMPN	326
LDB	135
*LOOKUP	076
†LPT	124
LSH	242
LSHC	246
†MAP	257
MOVE	200
MOVEI	201
MOVEM	202
MOVES	203
MOVM	214
MOVMI	215
MOVMM	216
MOVMS	217
MOVN	210
MOVNI	211
MOVNM	212
MOVNS	213
MOVS	204
MOVSI	205
MOVSM	206
MOVSS	207
*MTAPE	072
†MTC	220
†MTM	230
†MTS	224
MUL	224
MULB	227
MULI	225
MULM	226
*OPEN	050
OR	434
ORB	437
ORCA	454
ORCAB	457
ORCAI	455
ORCAM	456
ORCB	470
ORCBB	473
ORCBI	471
ORCBM	472
ORCM	464
ORCMB	467
ORCMI	465
ORCMM	466
ORI	435
ORM	436
*OUT	057
*OUTBUF	065
*OUTPUT	067
†PAG	010
†PI	004
†PLT	140
POP	262
POPJ	263
PORTAL	25404
†PTP	100
†PTR	104
PUSH	261
PUSHJ	260
*RELEAS	071
*RENAME	055
†RMC	270
ROT	241
ROTC	245
RSW	70004
SETA	424
SETAB	427
SETAI	425
SETAM	426
SETCA	450
SETCAB	453
SETCAI	451
SETCAM	452
SETCM	460
SETCMB	463
SETCMI	461
SETCMM	462
SETM	414
SETMB	417
SETMI	415
SETMM	416
SETO	474
SETOB	477
SETOI	475
SETOM	476
*SETSTS	060
SETZ	400
SETZB	403
SETZI	401
SETZM	402
SKIP	330
SKIPA	334
SKIPE	332
SKIPG	337
SKIPGE	335
SKIPL	331
SKIPLE	333
SKIPN	336
SOJ	360
SOJA	364
SOJE	362
SOJG	367
SOJGE	365
SOJL	361
SOJLE	363
SOJN	366
SOS	370
SOSA	374
SOSE	372
SOSG	377
SOSGE	375
SOSL	371
SOSLE	373
SOSN	376
*STATO	061
*STATUS	062
*STATZ	063
SUB	274
SUBB	277
SUBI	275
SUBM	276
TDC	650
TDCA	654
TDCE	652
TDCN	656
TDN	610
TDNA	614
TDNE	612
TDNN	616
TDO	670
TDOA	674
TDOE	672
TDON	676

FIRST OCTAL DIGIT \ SECOND OCTAL DIGIT	00	04	10	14	20	24	30	34	40	44	50	54	60	64	70	74
0	APR CPA CENTRAL PROCESSOR	PI PRIORITY INTERRUPT	PAG* KI10 PAGING	CCI PDP-8,9 INTERFACE	CCI2 PDP-8,9 INTERFACE	ADC ANALOG-DIGITAL CONVERTER	ADC2 ANALOG-DIGITAL CONVERTER						DLB PDP-11 DATA LINK	DLC	CLK REAL TIME CLOCK	CLK2 REAL TIME CLOCK
1	PTP PAPER TAPE PUNCH	PTR PAPER TAPE READER	CDP CARD PUNCH	CDR CARD READER	TTY CONSOLE TELETYPE	LPT LINE PRINTER	DIS DISPLAY	DIS2 DISPLAY	PLT PLOTTER	PLT2 PLOTTER	CR CARD READER	CR2 CARD READER	DLB2* PDP-11 DATA LINK	DLC2	DSK DISK/DRUM	DSK2 DISK/DRUM
2	DC DATA CONTROL	DC2 DATA CONTROL	UTC DECTAPE	UTS	MTC MAGNETIC TAPE	MTS	MTM†	LPT2† LINE PRINTER	DLS DATA LINE SCANNER	DLS2 DATA LINE SCANNER	DPC DISK PACK SYSTEM	DPC2 DISK PACK SYSTEM	DPC3 DISK PASK SYSTEM	DPC4 DISK PACK SYSTEM	RMC* DATA CONTROL	RMC2 DATA CONTROL
3	DCSA DATA COMMUNICATION	DCSB DATA COMMUNICATION			DTC DECTAPE	DTS DECTAPE	DTC2 DECTAPE	DTS2 DECTAPE	TMC MAGNETIC TAPE	TMS MAGNETIC TAPE	TMC2 MAGNETIC TAPE	TMS2 MAGNETIC TAPE				
4													DSS SINGLE SYNCHRONOUS LINE UNIT	DSI	DSS2 SINGLE SYNCHRONOUS LINE UNIT	DSI2
5																
6																
7																

CODES IN THIS SECTION RESERVED FOR USER SPECIAL DEVICES

KI10 UNRESTRICTED CODES RESERVED FOR USERS

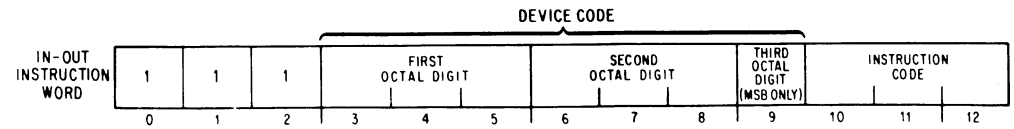
KI10 UNRESTRICTED CODES RESERVED FOR DEC

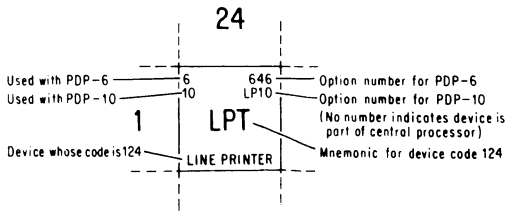
*IN THE PDP-6 THESE CODES ARE USED FOR OTHER DEVICES † FOR A THIRD LINE PRINTER USE CODE 230

010 DRUM PROCESSOR

160 PDP-7,8 INTERFACE

270 DISK FILE (DF)





DEVICE MNEMONICS

ALGEBRAIC REPRESENTATION

The remaining pages of this Appendix list, in symbolic form, the actual operations performed by the instructions. The grouping, as given below, differs slightly from that used in Chapter 2.

The terminology and notation used also vary somewhat from that in the body of the manual, as follows.

AC — The accumulator address in bits 9–12 of the instruction word (represented by *A* in the instruction descriptions).

AC+1 — The address one greater than AC, except that AC+1 is 0 if AC is 17.

E — The result of the effective address calculation. E is eighteen bits when used as an address, half word operand, mask or output conditions, but is a signed 9-bit quantity when used as a scale factor or a shift number.

E+1 — The address one greater than E, except that E+1 is 0 if E is 777777.

PC — The 18-bit program counter.

(X) — The word contained in register X.

$(X)_L$ — The left half of (X).

$(X)_R$ — The right half of (X).

$(X)_S$ — The word contained in X with its left and right halves swapped.

A_n — The value of bit n of the quantity A.

A,B — A 36-bit word with the 18-bit quantity A in its left half and the 18-bit quantity B in its right half (either A or B may be 0).

(X,Y) — The contents of registers X and Y concatenated into a double word operand.

$((X))$ — The word contained in the register addressed by (X), *ie* addressed by the word in register X.

$A \rightarrow B$ — The quantity A replaces the quantity B (A and B may be half words, full words or double words). *Eg*

$$(AC) + (E) \rightarrow (AC)$$

means the word in accumulator AC plus the word in memory location E replaces the word in AC.

(AC) (E) — The word in AC and the word in E.

$\wedge \vee \forall \sim$ — The Boolean operators AND, inclusive OR, exclusive OR, and complement (logical negation).

$+ - \times \div \; ||$ The arithmetic operators for addition, negation or subtraction, multiplication, division, and absolute value (magnitude).

Square brackets are used occasionally for grouping. With respect to the values of their terms, the equations for a given instruction are in chronological order; *eg* in the pair of equations

$$(AC) + 1 \rightarrow (AC)$$
$$If\ (AC) = 0:\ E \rightarrow (PC)$$

the quantity tested in the second equation is the word in AC after it has been incremented by one.

Boolean

SETZ	400	0 → (AC)	SETO	474	777777777777 → (AC)
SETZI	401	0 → (AC)	SETOI	475	777777777777 → (AC)
SETZM	402	0 → (E)	SETOM	476	777777777777 → (E)
SETZB	403	0 → (AC) (E)	SETOB	477	777777777777 → (AC) (E)
SETA	424	(AC) → (AC) [*no-op*]	SETCA	450	~ (AC) → (AC)
SETAI	425	(AC) → (AC) [*no-op*]	SETCAI	451	~ (AC) → (AC)
SETAM	426	(AC) → (E)	SETCAM	452	~ (AC) → (E)
SETAB	427	(AC) → (E)	SETCAB	453	~ (AC) → (AC) (E)
SETM	414	(E) → (AC)	SETCM	460	~ (E) → (AC)
SETMI	415	0,E → (AC)	SETCMI	461	~ [0,E] → (AC)
SETMM	416	(E) → (E) [*no-op*]	SETCMM	462	~ (E) → (E)
SETMB	417	(E) → (AC) (E)	SETCMB	463	~ (E) → (AC) (E)
AND	404	(AC) ∧ (E) → (AC)	ANDCA	410	~ (AC) ∧ (E) → (AC)
ANDI	405	(AC) ∧ 0,E → (AC)	ANDCAI	411	~ (AC) ∧ 0,E → (AC)
ANDM	406	(AC) ∧ (E) → (E)	ANDCAM	412	~ (AC) ∧ (E) → (E)
ANDB	407	(AC) ∧ (E) → (AC) (E)	ANDCAB	413	~ (AC) ∧ (E) → (AC) (E)
ANDCM	420	(AC) ∧ ~ (E) → (AC)	ANDCB	440	~ (AC) ∧ ~ (E) → (AC)
ANDCMI	421	(AC) ∧ ~ [0,E] → (AC)	ANDCBI	441	~ (AC) ∧ ~ [0,E] → (AC)
ANDCMM	422	(AC) ∧ ~ (E) → (E)	ANDCBM	442	~ (AC) ∧ ~ (E) → (E)
ANDCMB	423	(AC) ∧ ~ (E) → (AC) (E)	ANDCBB	443	~ (AC) ∧ ~ (E) → (AC) (E)
IOR	434	(AC) ∨ (E) → (AC)	ORCA	454	~ (AC) ∨ (E) → (AC)
IORI	435	(AC) ∨ 0,E → (AC)	ORCAI	455	~ (AC) ∨ 0,E → (AC)
IORM	436	(AC) ∨ (E) → (E)	ORCAM	456	~ (AC) ∨ (E) → (E)
IORB	437	(AC) ∨ (E) → (AC) (E)	ORCAB	457	~ (AC) ∨ (E) → (AC) (E)
ORCM	464	(AC) ∨ ~ (E) → (AC)	ORCB	470	~ (AC) ∨ ~ (E) → (AC)
ORCMI	465	(AC) ∨ ~ [0,E] → (AC)	ORCBI	471	~ (AC) ∨ ~ [0,E] → (AC)
ORCMM	466	(AC) ∨ ~ (E) → (E)	ORCBM	472	~ (AC) ∨ ~ (E) → (E)
ORCMB	467	(AC) ∨ ~ (E) → (AC) (E)	ORCBB	473	~ (AC) ∨ ~ (E) → (AC) (E)
XOR	430	(AC) ∀ (E) → (AC)	EQV	444	~ [(AC) ∀ (E)] → (AC)
XORI	431	(AC) ∀ 0,E → (AC)	EQVI	445	~ [(AC) ∀ 0,E] → (AC)
XORM	432	(AC) ∀ (E) → (E)	EQVM	446	~ [(AC) ∀ (E)] → (E)
XORB	433	(AC) ∀ (E) → (AC) (E)	EQVB	447	~ [(AC) ∀ (E)] → (AC) (E)

Byte Manipulation

IBP	133	*Operations on* (E) [*see page 2-16*] *If* P − S ⩾ 0: P − S → P *If* P − S < 0: Y + 1 → Y 36 − S → P
LDB	135	BYTE IN ((E)) → (AC) [*see page 2-16*]
DPB	137	BYTE IN (AC) → BYTE IN ((E)) [*see page 2-16*]
ILDB	134	IBP *and* LDB
IDPB	136	IBP *and* DPB

Fixed Point Arithmetic

ADD	270	(AC) + (E) → (AC)	SUB	274	(AC) − (E) → (AC)
ADDI	271	(AC) + 0,E → (AC)	SUBI	275	(AC) − 0,E → (AC)
ADDM	272	(AC) + (E) → (E)	SUBM	276	(AC) − (E) → (E)
ADDB	273	(AC) + (E) → (AC) (E)	SUBB	277	(AC) − (E) → (AC) (E)
IMUL	220	(AC) × (E) → (AC)*	MUL	224	(AC) × (E) → (AC,AC+1)
IMULI	221	(AC) × 0,E → (AC)*	MULI	225	(AC) × 0,E → (AC,AC+1)
IMULM	222	(AC) × (E) → (E)*	MULM	226	(AC) × (E) → (E)†
IMULB	223	(AC) × (E) → (AC) (E)*	MULB	227	(AC) × (E) → (AC,AC+1) (E)
IDIV	230	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	DIV	234	(AC,AC+1) ÷ (E) → (AC) REMAINDER → (AC+1)
IDIVI	231	(AC) ÷ 0,E → (AC) REMAINDER → (AC+1)	DIVI	235	(AC,AC+1) ÷ 0,E → (AC) REMAINDER → (AC+1)
IDIVM	232	(AC) ÷ (E) → (E)	DIVM	236	(AC,AC+1) ÷ (E) → (E)
IDIVB	233	(AC) ÷ (E) → (AC) (E) REMAINDER → (AC+1)	DIVB	237	(AC,AC+1) ÷ (E) → (AC) (E) REMAINDER → (AC+1)

*The high order word of the product is discarded.
†The low order word of the product is discarded.

Floating Point Arithmetic

FAD	140	(AC) + (E) → (AC)	FADR	144	(AC) + (E) → (AC)
FADL	141	(AC) + (E) → (AC,AC+1)	FADRI	145	(AC) + E,0 → (AC)
FADM	142	(AC) + (E) → (E)	FADRM	146	(AC) + (E) → (E)
FADB	143	(AC) + (E) → (AC) (E)	FADRB	147	(AC) + (E) → (AC) (E)
FSB	150	(AC) − (E) → (AC)	FSBR	154	(AC) − (E) → (AC)
FSBL	151	(AC) − (E) → (AC,AC+1)	FSBRI	155	(AC) − E,0 → (AC)
FSBM	152	(AC) − (E) → (E)	FSBRM	156	(AC) − (E) → (E)
FSBB	153	(AC) − (E) → (AC) (E)	FSBRB	157	(AC) − (E) → (AC) (E)

FMP	160	(AC) X (E) → (AC)	FMPR	164	(AC) X (E) → (AC)
FMPL	161	(AC) X (E) → (AC,AC+1)	FMPRI	165	(AC) X E,0 → (AC)
FMPM	162	(AC) X (E) → (E)	FMPRM	166	(AC) X (E) → (E)
FMPB	163	(AC) X (E) → (AC) (E)	FMPRB	167	(AC) X (E) → (AC) (E)
FDV	170	(AC) ÷ (E) → (AC)	FDVR	174	(AC) ÷ (E) → (AC)
FDVL	171	(AC) ÷ (E) → (AC) REMAINDER → (AC+1)	FDVRI	175	(AC) ÷ E,0 → (AC)
FDVM	172	(AC) ÷ (E) → (E)	FDVRM	176	(AC) ÷ (E) → (E)
FDVB	173	(AC) ÷ (E) → (AC) (E)	FDVRB	177	(AC) ÷ (E) → (AC) (E)

UFA	130	(AC) + (E) → (AC+1) *without normalization*
DFN	131	− (AC,E) → (AC,E)
FSC	132	(AC) X 2^E → (AC)
FLTR	127	(E) *floated, rounded* → (AC)

FIX	122	(E) *fixed* → (AC)	FIXR	126	(E) *fixed, rounded* → (AC)

DFAD	110	(AC,AC+1) + (E,E+1) → (AC,AC+1)
DFSB	111	(AC,AC+1) − (E,E+1) → (AC,AC+1)
DFMP	112	(AC,AC+1) X (E,E+1) → (AC,AC+1)
DFDV	113	(AC,AC+1) ÷ (E,E+1) → (AC,AC+1)

DMOVE	120	(E,E+1) → (AC,AC+1)	DMOVEM	124	(AC,AC+1) → (E,E+1)
DMOVN	121	− (E,E+1) → (AC,AC+1)	DMOVNM	125	− (AC,AC+1) → (E,E+1)

Full Word Data Transmission

EXCH	250	(AC) ↔ (E)
BLT	251	*Move* E − $(AC)_R$ + 1 *words starting with* $((AC)_L)$ → $((AC)_R)$ [*see page 2-10*]

MOVE	200	(E) → (AC)	MOVS	204	$(E)_S$ → (AC)
MOVEI	201	0,E → (AC)	MOVSI	205	E,0 → (AC)
MOVEM	202	(AC) → (E)	MOVSM	206	$(AC)_S$ → (E)
MOVES	203	*If* AC ≠ 0: (E) → (AC)	MOVSS	207	$(E)_S$ → (E) *If* AC ≠ 0: (E) → (AC)
MOVN	210	− (E) → (AC)	MOVM	214	|(E)| → (AC)
MOVNI	211	− [0,E] → (AC)	MOVMI	215	0,E → (AC)
MOVNM	212	− (AC) → (E)	MOVMM	216	|(AC)| → (E)
MOVNS	213	− (E) → (E) *If* AC ≠ 0: (E) → (AC)	MOVMS	217	|(E)| → (E) *If* AC ≠ 0: (E) → (AC)

Half Word Data Transmission

HLL	500	$(E)_L \rightarrow (AC)_L$	HLLZ	510	$(E)_L, 0 \rightarrow (AC)$
HLLI	501	$0 \rightarrow (AC)_L$	HLLZI	511	$0 \rightarrow (AC)$
HLLM	502	$(AC)_L \rightarrow (E)_L$	HLLZM	512	$(AC)_L, 0 \rightarrow (E)$
HLLS	503	*If* AC ≠ 0: $(E) \rightarrow (AC)$	HLLZS	513	$0 \rightarrow (E)_R$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HLLO	520	$(E)_L, 777777 \rightarrow (AC)$	HLLE	530	$(E)_L, [(E)_0 \times 777777] \rightarrow (AC)$
HLLOI	521	$0, 777777 \rightarrow (AC)$	HLLEI	531	$0 \rightarrow (AC)$
HLLOM	522	$(AC)_L, 777777 \rightarrow (E)$	HLLEM	532	$(AC)_L, [(AC)_0 \times 777777] \rightarrow (E)$
HLLOS	523	$777777 \rightarrow (E)_R$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HLLES	533	$(E)_0 \times 777777 \rightarrow (E)_R$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HLR	544	$(E)_L \rightarrow (AC)_R$	HLRZ	554	$0, (E)_L \rightarrow (AC)$
HLRI	545	$0 \rightarrow (AC)_R$	HLRZI	555	$0 \rightarrow (AC)$
HLRM	546	$(AC)_L \rightarrow (E)_R$	HLRZM	556	$0, (AC)_L \rightarrow (E)$
HLRS	547	$(E)_L \rightarrow (E)_R$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HLRZS	557	$0, (E)_L \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HLRO	564	$777777, (E)_L \rightarrow (AC)$	HLRE	574	$[(E)_0 \times 777777], (E)_L \rightarrow (AC)$
HLROI	565	$777777, 0 \rightarrow (AC)$	HLREI	575	$0 \rightarrow (AC)$
HLROM	566	$777777, (AC)_L \rightarrow (E)$	HLREM	576	$[(AC)_0 \times 777777], (AC)_L \rightarrow (E)$
HLROS	567	$777777, (E)_L \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HLRES	577	$[(E)_0 \times 777777], (E)_L \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HRR	540	$(E)_R \rightarrow (AC)_R$	HRRZ	550	$0, (E)_R \rightarrow (AC)$
HRRI	541	$E \rightarrow (AC)_R$	HRRZI	551	$0, E \rightarrow (AC)$
HRRM	542	$(AC)_R \rightarrow (E)_R$	HRRZM	552	$0, (AC)_R \rightarrow (E)$
HRRS	543	*If* AC ≠ 0: $(E) \rightarrow (AC)$	HRRZS	553	$0 \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HRRO	560	$777777, (E)_R \rightarrow (AC)$	HRRE	570	$[(E)_{18} \times 777777], (E)_R \rightarrow (AC)$
HRROI	561	$777777, E \rightarrow (AC)$	HRREI	571	$[E_{18} \times 777777], E \rightarrow (AC)$
HRROM	562	$777777, (AC)_R \rightarrow (E)$	HRREM	572	$[(AC)_{18} \times 777777], (AC)_R \rightarrow (E)$
HRROS	563	$777777 \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HRRES	573	$(E)_{18} \times 777777 \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$
HRL	504	$(E)_R \rightarrow (AC)_L$	HRLZ	514	$(E)_R, 0 \rightarrow (AC)$
HRLI	505	$E \rightarrow (AC)_L$	HRLZI	515	$E, 0 \rightarrow (AC)$
HRLM	506	$(AC)_R \rightarrow (E)_L$	HRLZM	516	$(AC)_R, 0 \rightarrow (E)$
HRLS	507	$(E)_R \rightarrow (E)_L$ *If* AC ≠ 0: $(E) \rightarrow (AC)$	HRLZS	517	$(E)_R, 0 \rightarrow (E)$ *If* AC ≠ 0: $(E) \rightarrow (AC)$

HRLO	524	$(E)_R$,777777 → (AC)	HRLE	534	$(E)_R$,[$(E)_{18}$ X 777777] → (AC)
HRLOI	525	E,777777 → (AC)	HRLEI	535	E,[E_{18} X 777777] → (AC)
HRLOM	526	$(AC)_R$,777777 → (E)	HRLEM	536	$(AC)_R$,[$(AC)_{18}$ X 777777] → (E)
HRLOS	527	$(E)_R$,777777 → (E) *If* AC ≠ 0: (E) → (AC)	HRLES	537	$(E)_R$,[$(E)_{18}$ X 777777] → (E) *If* AC ≠ 0: (E) → (AC)

In-out

CONO	70020	E → COMMAND	CONSZ	70030	*If* $STATUS_R$ ∧ E = 0: *skip*
CONI	70024	STATUS → (E)	CONSO	70034	*If* $STATUS_R$ ∧ E ≠ 0: *skip*
DATAO	70014	(E) → DATA	DATAI	70004	DATA → (E)
BLKO	70010	(E) + 1000001 → (E)*	$((E)_R)$ → DATA [*see page 2-83*]		
BLKI	70000	(E) + 1000001 → (E)*	DATA → $((E)_R)$ [*see page 2-83*]		

Program Control

JSR	264	FLAGS,(PC) → (E)	E + 1 → (PC)	
JSP	265	FLAGS,(PC) → (AC)	E → (PC)	
JRST	254	E → (PC)	[*If* AC ≠ 0, *see page 2-63*]	
JSA	266	(AC) → (E)	E,(PC) → (AC)	E + 1 → (PC)
JRA	267	E → (PC)	$((AC)_L)$ → (AC)	
JFCL	255	*If* AC ∧ FLAGS ≠ 0:	E → (PC)	~ AC ∧ FLAGS → FLAGS
XCT	256	*Execute* (E)		
JFFO	243	*If* (AC) = 0: 0 → (AC + 1) *If* (AC) ≠ 0: E → (PC) [*see page 2-61*]		
MAP	257	PHYSICAL MAP DATA → (AC)		

Pushdown List

PUSH	261	(AC) + 1000001 → (AC)*	(E) → $((AC)_R)$	
POP	262	$((AC)_R)$ → (E)	(AC) − 1000001 → (AC)*	
PUSHJ	260	(AC) + 1000001 → (AC)*	FLAGS,(PC) → $((AC)_R)$	E → (PC)
POPJ	263	$((AC)_R)_R$ → (PC)	(AC) − 1000001 → (AC)*	

Shift and Rotate

ASH	240	(AC) X 2^E → (AC)	ASHC	244	(AC,AC+1) X 2^E → (AC,AC+1)
ROT	241	*Rotate* (AC) E *places*	ROTC	245	*Rotate* (AC,AC+1) E *places*
LSH	242	*Shift* (AC) E *places*	LSHC	246	*Shift* (AC,AC+1) E *places*

*In the KI10, 1 is added to or subtracted from each half separately.

Arithmetic Testing

AOBJP	252	(AC) + 1000001 → (AC)*	*If* (AC) ⩾ 0: E → (PC)
AOBJN	253	(AC) + 1000001 → (AC)*	*If* (AC) < 0: E → (PC)

CAI	300	*No-op*	CAM	310	*No-op*
CAIL	301	*If* (AC) < E: *skip*	CAML	311	*If* (AC) < (E): *skip*
CAIE	302	*If* (AC) = E: *skip*	CAME	312	*If* (AC) = (E): *skip*
CAILE	303	*If* (AC) ⩽ E: *skip*	CAMLE	313	*If* (AC) ⩽ (E): *skip*
CAIA	304	*Skip*	CAMA	314	*Skip*
CAIGE	305	*If* (AC) ⩾ E: *skip*	CAMGE	315	*If* (AC) ⩾ (E): *skip*
CAIN	306	*If* (AC) ≠ E: *skip*	CAMN	316	*If* (AC) ≠ (E): *skip*
CAIG	307	*If* (AC) > E: *skip*	CAMG	317	*If* (AC) > (E): *skip*
JUMP	320	*No-op*	SKIP	330	*If* AC ≠ 0: (E) → (AC)
JUMPL	321	*If* (AC) < 0: E → (PC)	SKIPL	331	*If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*
JUMPE	322	*If* (AC) = 0: E → (PC)	SKIPE	332	*If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*
JUMPLE	323	*If* (AC) ⩽ 0: E → (PC)	SKIPLE	333	*If* AC ≠ 0: (E) → (AC) *If* (E) ⩽ 0: *skip*
JUMPA	324	E → (PC)	SKIPA	334	*If* AC ≠ 0: (E) → (AC) *Skip*
JUMPGE	325	*If* (AC) ⩾ 0: E → (PC)	SKIPGE	335	*If* AC ≠ 0: (E) → (AC) *If* (E) ⩾ 0: *skip*
JUMPN	326	*If* (AC) ≠ 0: E → (PC)	SKIPN	336	*If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
JUMPG	327	*If* (AC) > 0: E → (PC)	SKIPG	337	*If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*
AOJ	340	(AC) + 1 → (AC)	SOJ	360	(AC) − 1 → (AC)
AOJL	341	(AC) + 1 → (AC) *If* (AC) < 0: E → (PC)	SOJL	361	(AC) − 1 → (AC) *If* (AC) < 0: E → (PC)
AOJE	342	(AC) + 1 → (AC) *If* (AC) = 0: E → (PC)	SOJE	362	(AC) − 1 → (AC) *If* (AC) = 0: E → (PC)
AOJLE	343	(AC) + 1 → (AC) *If* (AC) ⩽ 0: E → (PC)	SOJLE	363	(AC) − 1 → (AC) *If* (AC) ⩽ 0: E → (PC)
AOJA	344	(AC) + 1 → (AC) E → (PC)	SOJA	364	(AC) − 1 → (AC) E → (PC)
AOJGE	345	(AC) + 1 → (AC) *If* (AC) ⩾ 0: E → (PC)	SOJGE	365	(AC) − 1 → (AC) *If* (AC) ⩾ 0: E → (PC)

*In the KI10, 1 is added to or subtracted from each half separately.

AOJN	346	(AC) + 1 → (AC) *If* (AC) ≠ 0: E → (PC)	SOJN	366	(AC) − 1 → (AC) *If* (AC) ≠ 0: E → (PC)
AOJG	347	(AC) + 1 → (AC) *If* (AC) > 0: E → (PC)	SOJG	367	(AC) − 1 → (AC) *If* (AC) > 0: E → (PC)
AOS	350	(E) + 1 → (E) *If* (AC) ≠ 0: (E) → (AC)	SOS	370	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC)
AOSL	351	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*	SOSL	371	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) < 0: *skip*
AOSE	352	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*	SOSE	372	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) = 0: *skip*
AOSLE	353	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ⩽ 0: *skip*	SOSLE	373	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ⩽ 0: *skip*
AOSA	354	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *Skip*	SOSA	374	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *Skip*
AOSGE	355	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ⩾ 0: *skip*	SOSGE	375	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ⩾ 0: *skip*
AOSN	356	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*	SOSN	376	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) ≠ 0: *skip*
AOSG	357	(E) + 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*	SOSG	377	(E) − 1 → (E) *If* AC ≠ 0: (E) → (AC) *If* (E) > 0: *skip*

Logical Testing and Modification

TLN	601	*No-op*	TRN	600	*No-op*
TLNE	603	*If* $(AC)_L \wedge E = 0$: *skip*	TRNE	602	*If* $(AC)_R \wedge E = 0$: *skip*
TLNA	605	*Skip*	TRNA	604	*Skip*
TLNN	607	*If* $(AC)_L \wedge E \neq 0$: *skip*	TRNN	606	*If* $(AC)_R \wedge E \neq 0$: *skip*
TLZ	621	$(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZ	620	$(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZE	623	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZE	622	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$
TLZA	625	$(AC)_L \wedge \sim E \rightarrow (AC)_L$ *skip*	TRZA	624	$(AC)_R \wedge \sim E \rightarrow (AC)_R$ *skip*
TLZN	627	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \wedge \sim E \rightarrow (AC)_L$	TRZN	626	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \wedge \sim E \rightarrow (AC)_R$

TLC	641	$(AC)_L \forall E \rightarrow (AC)_L$	TRC	640	$(AC)_R \forall E \rightarrow (AC)_R$
TLCE	643	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \forall E \rightarrow (AC)_L$	TRCE	642	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \forall E \rightarrow (AC)_R$
TLCA	645	$(AC)_L \forall E \rightarrow (AC)_L$ *skip*	TRCA	644	$(AC)_R \forall E \rightarrow (AC)_R$ *skip*
TLCN	647	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \forall E \rightarrow (AC)_L$	TRCN	646	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \forall E \rightarrow (AC)_R$
TLO	661	$(AC)_L \vee E \rightarrow (AC)_L$	TRO	660	$(AC)_R \vee E \rightarrow (AC)_R$
TLOE	663	*If* $(AC)_L \wedge E = 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TROE	662	*If* $(AC)_R \wedge E = 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TLOA	665	$(AC)_L \vee E \rightarrow (AC)_L$ *skip*	TROA	664	$(AC)_R \vee E \rightarrow (AC)_R$ *skip*
TLON	667	*If* $(AC)_L \wedge E \neq 0$: *skip* $(AC)_L \vee E \rightarrow (AC)_L$	TRON	666	*If* $(AC)_R \wedge E \neq 0$: *skip* $(AC)_R \vee E \rightarrow (AC)_R$
TDN	610	*No-op*	TSN	611	*No-op*
TDNE	612	*If* $(AC) \wedge (E) = 0$: *skip*	TSNE	613	*If* $(AC) \wedge (E)_S = 0$: *skip*
TDNA	614	*Skip*	TSNA	615	*Skip*
TDNN	616	*If* $(AC) \wedge (E) \neq 0$: *skip*	TSNN	617	*If* $(AC) \wedge (E)_S \neq 0$: *skip*
TDZ	630	$(AC) \wedge \sim (E) \rightarrow (AC)$	TSZ	631	$(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZE	632	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZE	633	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDZA	634	$(AC) \wedge \sim (E) \rightarrow (AC)$ *skip*	TSZA	635	$(AC) \wedge \sim (E)_S \rightarrow (AC)$ *skip*
TDZN	636	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \wedge \sim (E) \rightarrow (AC)$	TSZN	637	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \wedge \sim (E)_S \rightarrow (AC)$
TDC	650	$(AC) \forall (E) \rightarrow (AC)$	TSC	651	$(AC) \forall (E)_S \rightarrow (AC)$
TDCE	652	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \forall (E) \rightarrow (AC)$	TSCE	653	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \forall (E)_S \rightarrow (AC)$
TDCA	654	$(AC) \forall (E) \rightarrow (AC)$ *skip*	TSCA	655	$(AC) \forall (E)_S \rightarrow (AC)$ *skip*
TDCN	656	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \forall (E) \rightarrow (AC)$	TSCN	657	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \forall (E)_S \rightarrow (AC)$
TDO	670	$(AC) \vee (E) \rightarrow (AC)$	TSO	671	$(AC) \vee (E)_S \rightarrow (AC)$
TDOE	672	*If* $(AC) \wedge (E) = 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSOE	673	*If* $(AC) \wedge (E)_S = 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$
TDOA	674	$(AC) \vee (E) \rightarrow (AC)$ *skip*	TSOA	675	$(AC) \vee (E)_S \rightarrow (AC)$ *skip*
TDON	676	*If* $(AC) \wedge (E) \neq 0$: *skip* $(AC) \vee (E) \rightarrow (AC)$	TSON	677	*If* $(AC) \wedge (E)_S \neq 0$: *skip* $(AC) \vee (E)_S \rightarrow (AC)$

POWERS OF TWO

2^N	N	2^{-N}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 333 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

APPENDIX B

INPUT-OUTPUT CODES

The table beginning on the next page lists the complete 1968 ASCII code (ANSI X3.4-1968). The software handles the full character set, and for a program that does not handle lower case, it translates input codes 140–174 into the corresponding upper case codes (100–134) and translates both 175 and 176 into 033, escape. The actual character sets available on different terminals vary greatly, but usually a terminal without lower case will accept lower case codes, printing the corresponding upper case character. The definitions of the control codes are those given by ASCII; most control codes, however, have no effect on the console terminal, and the definitions bear no necessary relation to the use of the codes in conjunction with the DECsystem–10 software. Brackets enclose earlier definitions of control codes (mostly 1963 ASCII). The table includes bit 8 as an even parity bit, the form generally used for paper tape and asynchronous operations; odd parity is generally used for magnetic tape and synchronous operations.

Output codes are simply passed on to the terminal as they are, with the expectation that the terminal will ignore irrevelant control codes, and that a terminal that lacks lower case will print the corresponding upper case. A terminal that fails to live up to these assumptions will generally not operate satisfactorily with the DECsystem–10 software.

With all line printers, ten control characters are used for format control, and the interface also recognizes null for fill and delete for selecting hidden characters. The 64-character print set includes the figures and upper case; lower case is added for the 96-character set (with the smaller print set, giving a lower case code prints the upper case character). The larger print set includes a character hidden under delete, a feature that is optional on the LP10D and H. The printable characters are generally those defined by ASCII, with little if any variation. The 128-character printer uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control codes that affect the printer and also under null and delete.

The first two pages of the table of card codes [*pages B-8 to B-12*] list the column punches required to represent characters in the ASCII card code. When reading cards, the software translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. There are also a few control hole patterns that the software responds to but does not translate. The next page lists two earlier DEC card codes that have only the figure and upper case character subset, plus a few control punches. The remaining pages of the table show the relationship among the early DEC card codes, the corresponding characters in the ASCII set, and several IBM card punches. Each column punch is produced by a single key on any keypunch for which a character is listed, the character being that which is printed at the top of the card.

ASCII CODE

Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Remarks
0	000	000	NUL	Null, tape feed. Control shift P.
1	001	001	SOH	Start of heading [SOM, start of message]. Control A.
1	002	002	STX	Start of text [EOA, end of address]. Control B.
0	003	003	ETX	End of text [EOM; end of message]. Control C.
1	004	004	EOT	End of transmission; shuts off TWX machines and disconnects some data sets. Control D.
0	005	005	ENQ	Enquiry [WRU, "Who are you?"]. Triggers identification ("Here is . . .") at remote station if so equipped. Control E.
0	006	006	ACK	Acknowledge [RU, "Are you . . . ?"]. Control F.
1	007	007	BEL	Rings the bell. Control G.
1	008	010	BS	Backspace. Control H.
0	009	011	HT	Horizontal tab. Control I.
0	010	012	LF	Line feed. Control J.
1	011	013	VT	Vertical tab. Control K.
0	012	014	FF	Form feed to top of next page. Control L.
1	013	015	CR	Carriage return to beginning of line. Control M.
1	014	016	SO	Shift out; change character set or change ribbon color to red. Control N.
0	015	017	SI	Shift in; return to standard character set or color. Control O.
1	016	020	DLE	Data link escape [DCO]. Control P.
0	017	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	018	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	019	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	020	024	DC4	Device control 4 (stop), turns punch or auxiliary off. Control T (~~TAPE~~, AUX OFF).
1	021	025	NAK	Negative acknowledge [ERR, error]. Control U.
1	022	026	SYN	Synchronous idle [SYNC]. Control V.
0	023	027	ETB	End of transmission block [LEM, logical end of medium]. Control W.
0	024	030	CAN	Cancel [S_0]. Control X.
1	025	031	EM	End of medium [S_1]. Control Y.
1	026	032	SUB	Substitute [S_2]. Control Z.
0	027	033	ESC	Escape, prefix [S_3]. Control shift K.
1	028	034	FS	File separator [S_4]. Control shift L.
0	029	035	GS	Group separator [S_5]. Control shift M.
0	030	036	RS	Record separator [S_6]. Control shift N.
1	031	037	US	Unit separator [S_7]. Control shift O.

Figures				Upper Case				Lower Case			
Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character	Even Parity Bit	7-Bit Decimal	7-Bit Octal	Character[10]
1	032	040	SP[1]	1	064	100	@[4]	0	096	140	`[11]
0	033	041	!	0	065	101	A	1	097	141	a
0	034	042	"	0	066	102	B	1	098	142	b
1	035	043	#[2]	1	067	103	C	0	099	143	c
0	036	044	$	0	068	104	D	1	100	144	d
1	037	045	%	1	069	105	E	0	101	145	e
1	038	046	&	1	070	106	F	0	102	146	f
0	039	047	'[3]	0	071	107	G	1	103	147	g
0	040	050	(	0	072	110	H	1	104	150	h
1	041	051	)	1	073	111	I	0	105	151	i
1	042	052	*	1	074	112	J	0	106	152	j
0	043	053	+	0	075	113	K	1	107	153	k
1	044	054	,	1	076	114	L	0	108	154	l
0	045	055	-	0	077	115	M	1	109	155	m
0	046	056	.	0	078	116	N	1	110	156	n
1	047	057	/	1	079	117	O	0	111	157	o
0	048	060	Ø	0	080	120	P	1	112	160	p
1	049	061	1	1	081	121	Q	0	113	161	q
1	050	062	2	1	082	122	R	0	114	162	r
0	051	063	3	0	083	123	S	1	115	163	s
1	052	064	4	1	084	124	T	0	116	164	t
0	053	065	5	0	085	125	U	1	117	165	u
0	054	066	6	0	086	126	V	1	118	166	v
1	055	067	7	1	087	127	W	0	119	167	w
1	056	070	8	1	088	130	X	0	120	170	x
0	057	071	9	0	089	131	Y	1	121	171	y
0	058	072	:	0	090	132	Z	1	122	172	z
1	059	073	;	1	091	133	[[5]	0	123	173	{
0	060	074	<	0	092	134	\[6]	1	124	174	¦[12]
1	061	075	=	1	093	135	][7]	0	125	175	}[13]
1	062	076	>	1	094	136	^[8]	0	126	176	~[14]
0	063	077	?	0	095	137	_[9]	1	127	177	DEL[15]

[1] Space.

[2] £ on some (non-DEC) units.

[3] Accent acute or apostrophe – ' before 1965, but used until recently on DEC units.

[4] ` 1965–67, but never on DEC units.

[5] Shift K.

[6] ~ 1965–67, but never on DEC units. Shift L.

[7] Shift M.

[8] Circumflex – ↑ before 1965, but used until recently on DEC units.

[9] Underscore – ← before 1965, but used until recently on DEC units.

[10] Codes 140–173 first defined in 1965. For a full ASCII character set the Monitor accepts codes 140–176 as lower case. For a character set that lacks lower case, the Monitor translates input codes 140–174 into the corresponding upper case codes (100–134) and translates both 175 and 176 into 033, escape. Early versions of the Monitor used 175 as the escape code and translated both 176 and 033 to it.

[11] Accent grave – @ 1965–67, but never on DEC units.

[12] Control character ACK before 1965; ¬ 1965–67, but never on DEC units. Vertical bar may or may not have gap depending on font design, but generally does not on DEC units.

[13] Unassigned control character (usually ALT MODE) before 1965. Code generated by ALT MODE key on most DEC units.

[14] Control character ESC before 1965; | 1965–67, but never on DEC units. Code generated by ESC key on some DEC units.

[15] Delete, rub out (not part of lower case set).

LINE PRINTER CODE: LP10A, B, C, D, E
Basic Character Set

Control				Figures				Upper Case			
Hex	Decimal	Octal	Character	Hex	Decimal	Octal	Character	Hex	Decimal	Octal	Character
09	009	011	HT	20	032	040	SP	40	064	100	@
0A	010	012	LF	21	033	041	!	41	065	101	A
0B	011	013	VT	22	034	042	"	42	066	102	B
0C	012	014	FF	23	035	043	#	43	067	103	C
0D	013	015	CR	24	036	044	$	44	068	104	D
				25	037	045	%	45	069	105	E
10	016	020	DLE	26	038	046	&	46	070	106	F
11	017	021	DC1	27	039	047	'	47	071	107	G
12	018	022	DC2	28	040	050	(	48	072	110	H
13	019	023	DC3	29	041	051	)	49	073	111	I
14	020	024	DC4	2A	042	052	*	4A	074	112	J
				2B	043	053	+	4B	075	113	K
00	000	000	NUL	2C	044	054	,	4C	076	114	L
7F	127	177	DEL	2D	045	055	-	4D	077	115	M
				2E	046	056	.	4E	078	116	N
				2F	047	057	/	4F	079	117	O
				30	048	060	0	50	080	120	P
				31	049	061	1	51	081	121	Q
				32	050	062	2	52	082	122	R
				33	051	063	3	53	083	123	S
				34	052	064	4	54	084	124	T
				35	053	065	5	55	085	125	U
				36	054	066	6	56	086	126	V
				37	055	067	7	57	087	127	W
				38	056	070	8	58	088	130	X
				39	057	071	9	59	089	131	Y
				3A	058	072	:	5A	090	132	Z
				3B	059	073	;	5B	091	133	[
				3C	060	074	<	5C	092	134	\
				3D	061	075	=	5D	093	135	]
				3E	062	076	>	5E	094	136	↑
				3F	063	077	?	5F	095	137	←

APPENDIX E

PROCESSOR COMPATIBILITY

The table beginning below identifies the programming differences among the various central processors. The reader is forewarned not to assume that he can program a new processor simply by glancing through this table. The simpler differences, principally those associated with individual user instructions, are explained adequately in the table entries. But in more complex cases, the table entries serve only to identify the area of difference and refer the reader to the real substance in Chapter 2. In particular, all programmers, regardless of previous experience with other processors, should read Chapter 1; and all system programmers should read the later material in Chapter 2 on interrupts, processor conditions, and program and memory management.

The table is limited to programming differences, and console switches are mentioned only insofar as they affect programming. Operating differences are so extensive, that upon approaching a new processor an operator must read the complete operating information given for it in Appendix F.

	PDP-6	KA10	KI10
Address Break	No.	Switch and flag – satisfaction of the address condition sets the flag, which is a processor condition and causes an interrupt [*refer to CONI APR,* §2.14 *and Appendix F2 for the address conditions*].	Switch but no flag – satisfaction of an address condition causes an address failure [*refer to Page Failure,* §2.15 *and Appendix F1 for the address conditions*].
Address Failure Inhibit	Not applicable.	Not applicable.	In PC word – see Address Break.
Address stop	Address switches are compared with virtual addresses (unrelocated).	Address switches are compared with physical addresses (relocated).	Address switches are compared with virtual addresses in space selected by paging switches [*Appendix F1*].
Auto restart	No.	No.	Yes [§2.14].
BLKI, BLKO *(also see IO instructions)*	Pointer is incremented by adding 1000001 to it.	Same as PDP-6.	The two halves of the pointer are incremented independently.
Byte pointer incrementing	Address overflow carries into index field, and effective address calculation from the pointer uses the newly specified index register.	Address overflow carries into index field, but effective address calculation from the pointer in an IDPB or ILDB uses the originally specified index register unless an inter-	Address overflow does not carry into index field.

	PDP-6	KA10	KI10
		rupt occurs between the two parts of the instruction, in which case the new index field is used and the result is as on the PDP-6.	
Carry flags	Subtraction is done in three steps: complement minuend, add, complement sum. The resulting effect on the Carry flags is the opposite of that listed for SUB, SOJ, SOS at the beginning of §2.9. *Eg* an SOS that decrements -2^{35} sets Carry 1 and has no effect on Carry 0.	Subtraction is done by directly adding the twos complement of the subtrahend, giving the Carry flag effects listed in §2.9.	Same as KA10.
Clock	Program must disable Clock interrupts when operator is using single instruction mode.	The flag is disabled while the single instruction switch is on.	Same as KA10.
Console programming	DATAI APR,.	DATAI APR, and DATAO PI, to load MI [§2.12].	Same as KA10 plus controls some switches with DATAO PTR, [§2.12] and reads switches in left half of CONI PI, [§2.13] and CONI APR, [§2.14].
D-A	Must have IO device.	Same as PDP-6.	Programmed with DATAO APR, [§2.14].
DFAD, DFSB, DFMP, DFDV	No.	No.	Yes.
DFN	No.	Yes.	Yes.
DMOVE, DMOVN	No.	No.	Yes.
FAD, FSB, FMP, FDV	*See floating point.*		
First Part Done	Set only by interrupt between parts of an IDPB or ILDB, and cleared only when the PC word is saved by an interrupt instruction or an instruction executed by a UUO. (Flag is often referred to as Byte Increment Suppression.)	Set same as PDP-6, but cleared whenever the PC word is saved.	Set by interrupt same as KA10, but also set by page failure in second part of IDPB, ILDB, DMOVEM, DMOVNM, or noninterrupt BLKO or BLKI. Cleared same as KA10.
FIX, FIXR	No.	No.	Yes.
Flags	*See JFCL, JRSTF, PC word, processor conditions, interrupt status, and individual flags.*		
Floating Overflow	No.	Yes – in PC word and processor conditions.	Yes – in PC word.

	PDP-6	KA10	KI10
Floating point	*(Also see overflow.)*		
Instructions [§2.6]	FSC plus four single precision standard operations, with and without rounding, in basic, Long, Memory and Both modes.	PDP-6 complement except Immediate replaces Long with rounding, plus UFA and DFN to facilitate software double precision operations.	KA10 complement plus number conversion (FIX, FIXR, FLTR) and hardware double precision (DMOVE, DMOVN, DMOVEM, DMOVNM, DFAD, DFSB, DFMP, DFDV).
Immediate mode	No.	Yes – operand *E*,0, but only with rounding (FADRI, FSBRI, FMPRI, FDVRI).	Same as KA10.
Long mode			
FADL, FSBL, FMPL	In low order word, fraction begins in bit 1 (no exponent), and sign is not forced to 0.	Low order word has fraction and exponent in standard software format described in §1.1.	Same as KA10.
With rounding	Stores meaningless low order word or remainder.	Replaced by immediate mode.	Same as KA10.
FDVL	Remainder is incorrect and lacks exponent.	Correct remainder is in standard floating point format but is not normalized.	Same as KA10.
Normalization	In add, subtract and multiply, if high order word of double length result is (positive) zero, no normalization takes place. Hence except in long mode, all bits of significance are lost. In FSC, hardware does not normalize result.	Result is always normalized (except of course in UFA and DFN, and remainders and low order words are never normalized separately).	Same as KA10.
Rounding	If low order part is exactly ½LSB in a negative number, rounding is toward zero (decreasing magnitude).	A low order part of exactly ½LSB is rounded away from zero (increasing magnitude).	Same as KA10.
Floating Underflow	No.	Yes – in PC word and processor conditions.	Yes – in PC word.
FLTR	No.	No.	Yes.
FMP, FSB	*See floating point.*		
FSC	Hardware does not normalize result.	Hardware does normalize result.	Same as KA10.
HALT	MA lights display address one greater than that containing instruction that caused halt. Cannot be performed in user mode.	Address display same as PDP-6. Can be performed in user mode only if User In-out set.	AR lights display address instead. Cannot be performed in user or supervisor mode.
IBP	*See byte pointer incrementing.*		
IDIV	Overflow if dividend -2^{35}.	Dividend -2^{35} OK except No Divide if divisor ±1.	Same as KA10.
IDPB, ILDB	*See byte pointer incrementing*		

	PDP-6	KA10	KI10
In-out Page Failure	Not applicable.	Not applicable.	In processor conditions.
Interrupt [§2.13]	Standard interrupt only.	Standard interrupt only.	Device returns function word that selects one of five interrupt functions described in §2.13.
Interrupt locations	Executive (physical) locations 42–57.	Executive locations 42–57, or 142–157 if offset.	Locations 42–57 in executive process table.
Interrupt instructions	Must use JSR to enter interrupt routine. Only a completed BLK*X* goes to second location. Only a DATA*X* or incomplete BLK*X* dismisses automatically. A condition IO instruction in first location or any IO instruction in second location hangs up processor.	Can use JSR, JSP, JSA, PUSHJ or JRST to enter interrupt routine. Otherwise same as PDP-6.	Can use JSR, JSP, PUSHJ, or MUUO to enter interrupt routine. Go to second location only when skip condition *not* satisfied in AOS*X*, SKIP*X*, SOS*X*, CONS*X* or BLK*X*. All other instructions dismiss automatically except that: In the second location a skip instruction whose condition is not satisfied hangs up the processor; LUUO, BLT, DMOVEM and DMOVNM will not work as interrupt instructions.
Interrupt points (besides between BLT transfers)	Before instruction fetch and each address word fetch.	After instruction fetch and each address word fetch.	After instruction done, after each address word fetch, in first half of DFDV, and when IO waiting for bus.
	The variation from one processor to another in allowable stopping points for an interrupt produces differences in the way the interrupt system responds to error situations (address break, memory protection violation, parity error, nonexistent memory). A common procedure is for the interrupt program, once it has recognized the error, to turn off the flag and get out of the processor channel quickly by switching over to a lower priority channel, dismissing the processor interrupt to the unmodified PC word. This works fine on the PDP-6 because it recognizes the lower priority interrupt before fetching the next instruction. But the greater complexity of the other processors leads to problems explained in the cautions that accompany the discussion of processor conditions in §2.14.		
Program initiated interrupts	Request dropped after interrupt.	Same as PDP-6.	Request stays on until turned off by program.
CONO PI.	22 Not used	22 Not used	22 Drop selected program requests
CONI PI.	Left half not used.	Left half not used.	Left half used for switches and program requests [§2.13].
	18 Power Failure	18 Power Failure	18 Not used
	19 Parity Error	19 Parity Error	19 Not used
	20 Parity Error Interrupt Enabled	20 Parity Error Interrupt Enabled	20 Not used
	21–27 Not used	21–27 Interrupts in progress	21–27 Interrupts in progress

	PDP-6	KA10	KI10
IO instructions	Can be performed in user mode only if User In-out set.	Same as PDP-6.	Cannot be performed in supervisor mode. Can be performed in user mode only with device codes 740 and above or if User In-out set.
JEN	Cannot be performed in user mode.	Can be performed in user mode only if User In-out set.	Cannot be performed in user or supervisor mode.
JFCL bit 12 (JFCL 1, JFOV)	PC change.	Floating Overflow.	Floating Overflow.
JFFO	No.	Yes.	Yes.
JRST 1,	Enter user mode.	Enter user mode.	PORTAL – clears Public when fetched from a nonpublic area, so is valid entry.
JRSTF (JRST 2,)	When used solely with indexing (no indirect), restores flags correctly only if previous instruction leaves left half of AR clear.	No problem.	No problem.
LUUO	*See UUO.*		
Maintenance programming	No.	No.	Yes [§2.14].
MAP	No.	No.	Yes [§2.15].
Memory management	One each, protection and relocation registers define user area. User illegal memory reference sets Memory Protection flag, a processor condition.	Two each, protection and relocation registers define a two-part user area where the high part can be write-protected [§2.16]. User illegal memory reference sets Memory Protection flag, a processor condition.	Paging hardware, where illegal memory reference causes page failure [§2.15].
Memory Protection interrupt	After an illegal user reference, the interrupt occurs before the next instruction fetch.	After an illegal user reference, the processor executes a zero instruction (UUO), which is trapped in executive location 40. The interrupt occurs after the instruction in executive location 41 is fetched.	Not applicable (page failures are trapped immediately).
Memory areas (= modes)	User (relocated) and executive (unrelocated).	Same as PDP-6.	User and executive areas divided into public and concealed areas distinguished by Public flag. User program execution thus in public or concealed mode; executive similarly in supervisor or kernel mode.

	PDP-6	KA10	KI10
Memory references	Unnecessary memory references are made in SETZ, SETO, SETA and SETCA. *Eg* SETZ AC,-1 fetches location 777777, which would likely be a non-existent memory reference or a protection violation.	The unnecessary references of the PDP-6 are not made.	Same as KA10.
	For memory reference instructions in which the mode configuration happens to produce a no-op – such as SETMM AC,M or SKIP 0,M or TDN AC,M – all machines make the reference even though it is unnecessary.		
MUL	AC supplies multiplicand, which if -2^{35} is treated as though it were $+2^{35}$.	Same as PDP-6.	AC supplies multiplier.
MUUO	*See UUO.*		
No Divide	No.	Yes – in PC word.	Same as KA10.
Overflow	Overflow (arithmetic) and Pushdown Overflow flags, which cause interrupts. Overflow conditions set flags in all circumstances.	Same as PDP-6 plus Floating Overflow, Floating Underflow and No Divide flags.	Same arithmetic flags as KA10 but no pushdown flag, and overflow handled by trapping instead of interrupts (arithmetic, Trap 1; pushdown, Trap 2). Overflow conditions set no flags in interrupt instructions.
PC Change	Yes.	No.	No.
PC word	0 Overflow	0 Overflow	0 Overflow in user mode, Disable Bypass in executive mode.
	3 PC Change	3 Floating Overflow	3 Floating Overflow
	7 Not used	7 Not used	7 Public
	8 Not used	8 Not used	8 Address Failure Inhibit
	9 Not used	9 Not used	9 Trap 2
	10 Not used	10 Not used	10 Trap 1
	11 Not used	11 Floating Underflow	11 Floating Underflow
	12 Not used	12 No Divide	12 No Divide
Processor conditions			
CONO APR,	18 Clear Pushdown Overflow	18 Clear Pushdown Overflow	18 Reset timer
	20 Not used	20 Not used	20 Disable timer
	21 Not used	21 Clear Address Break	21 Enable timer
	22 Clear Memory Protection	22 Clear Memory Protection	22 Disable auto restart
	23 Clear Nonexistent Memory	23 Clear Nonexistent Memory	23 Enable auto restart
	27 Disable PC Change Interrupt	27 Disable Floating Overflow Interrupt	27 Not used

	PDP-6	KA10	KI10
	28 Enable PC Change Interrupt	28 Enable Floating Overflow Interrupt	28 Clear In-out Page Failure
	29 Clear PC Change	29 Clear Floating Overflow	29 Clear Nonexistent Memory
	30 Disable Overflow Interrupt	30 Disable Overflow Interrupt	30–32 Processor PIA – error
	31 Enable Overflow Interrupt	31 Enable Overflow Interrupt	
	32 Clear Overflow	32 Clear Overflow	
	33–35 Processor PIA	33–35 Processor PIA	33–35 Processor PIA – clock
CONI APR,	Left half not used.	Left half not used.	Left half used for switches, etc [§2.14].
	18 Not used	18 Not used	18 Time Out
	19 Pushdown Overflow	19 Pushdown Overflow	19 Parity Error
	20 User In-out	20 User In-out	20 Parity Error Interrupt Enabled
	21 Not used	21 Address Break	21 Timer Enabled
	22 Memory Protection	22 Memory Protection	22 Power Failure
	23 Nonexistent Memory	23 Nonexistent Memory	23 Auto Restart Disabled
	28 PC Change Interrupt Enabled	28 Floating Overflow Interrupt Enabled	28 In-out Page Failure
	29 PC Change	29 Floating Overflow	29 Nonexistent Memory
	30 Not used	30 Trap Offset	30–32 Processor PIA – error
	31 Overflow Interrupt Enabled	31 Overflow Interrupt Enabled	
	32 Overflow	32 Overflow	
	33–35 Processor PIA	33–35 Processor PIA	33–35 Processor PIA – clock
DATAO APR,	No.	No.	Yes – maintenance and D-A [§2.14].
Parity Error	No.	Read by CONI PI,.	Read by CONI APR,.
POP, POPJ *(also see overflow)*.	Pointer is decremented by subtracting 1000001 from it.	Same as PDP-6.	The two halves of the pointer are decremented independently.
POP AC,AC	AC receives decremented pointer.	AC receives word taken from stack and pointer is lost.	Same as KA10.
Power Failure	No.	Read by CONI PI,.	Read by CONI APR,; also Power Alarm and auto restart feature [§2.14].
Processor serial number	Not available.	Not available.	Can be read by CONI PAG,.
Program management	User can never give HALT or JEN, can use IO only if User In-out set. Illegal instruction executes as UUO.	IO, HALT and JEN illegal unless User In-out set, in which case all are legal. Illegal instruction executes as MUUO.	User always has IO with codes 740–774, can use all IO if User In-out set. Supervisor can never use IO. Neither can ever give HALT or JEN.

	PDP-6	KA10	KI10
			Illegal instruction executes as MUUO.
Programmable margins	No.	No.	Yes [§2.14].
Public	Not applicable.	Not applicable.	In PC word; differentiates between public and concealed modes in memory management [§2.15].
PUSH, PUSHJ *(also see overflow)*	Pointer is incremented by adding 1000001 to it.	Same as PDP-6.	The two halves of the pointer are incremented independently.
Pushdown Overflow	*See overflow.*		
Read in	No hardware read in; key allows access to readin area (first 16 core locations) for bootstrap.	Yes. Ends by executing the last word in the block as an instruction [§2.12].	Yes. Selects kernel mode with executive paging disabled. Ends by jumping to the location containing the last word in the block [§2.12].
Shift-rotate	Shifts number of places specified by *E* (maximum 255).	Same as PDP-6.	Eliminates redundant shifting: Arithmetic or logical shift at most 72 places; Rotate *E* mod 72 places (except 72 places if *E* a nonzero multiple of 72).
SOJ, SOS	*See Carry flags.*		
Status	*See PC word, processor conditions, interrupt status.*		
SUB	*See Carry flags.*		
Timer	No.	No.	Yes – processor conditions [§2.14].
Trap flags, trapping	No (except UUO).	No (except UUO).	Yes – for arithmetic and pushdown overflow [§2.9], page failures [§2.15], and UUO; trap flags in PC word.
Trap Offset	No.	Turning on MA TRP OFFSET switch sets flag (a processor condition) and substitutes executive locations 140–161 for MUUO, interrupt and unassigned code locations 40–61 to distinguish between two processors using same memory.	Not applicable.
UFA	No.	Yes.	Yes.
Unassigned codes	100–131, 243, 247, 257. On most machines these execute like UUOs but use executive	100–127 execute like UUOs but use executive locations 60-61 (160-161 if offset).	100–107, 114–117, 123 and 247 execute like MUUOs.

	PDP-6	KA10	KI10
	locations 60-61; on some machines they execute as no-ops (there is no standard).	247 and 257 are not regarded as unassigned and execute as no-ops unless implemented by special hardware.	
Unimplemented operations	If floating point and byte instructions are not implemented in hardware, codes 132–177 act like unassigned codes.	Turning on FP TRP switch causes floating point and byte codes 130–177 to act like unassigned codes.	All assigned codes are implemented in hardware.
User In-out	Allows IO instructions to be performed in user mode. Flag is in PC word and processor conditions.	Allows IO instructions, HALT and JEN to be performed in user mode. Flag is in PC word and processor conditions.	Allows IO instructions with device codes under 740 to be performed in user mode. Also used to control special effects in executive XCT [§2.15]. Flag is in PC word only.
UUO	All UUOs 000–077 use executive locations 40-41.	LUUOs 001–037 use user locations 40-41 in user mode, executive locations 40-41 in executive mode. MUUOs 000 and 040–077 use executive locations 40-41. (Trap offset changes executive locations 40-41 to 140-141.)	LUUOs 001–037 use user locations 40-41 in user mode, locations 40-41 in executive process table in executive mode. MUUOs 000 and 040–077 store *CODE A,E* in location 424 of the executive process table, save the PC word in 425, and restart with the processor configured according to a new PC word [*for details refer to* §2.10].
XCT	Same in all program modes.	Same as PDP-6.	In executive mode, can be performed as executive XCT [§2.15].

Index

Modifications to the A.I. Lab PDP-6 and KA10

Appendix E of the Hardware Reference Manual describes the differences between the PDP-6, the KA10, and the KI10. This appendix describes the modifications that have been made to the A.I. Lab PDP-6 and KA10 processors.

PDP-6 modifications:

Opcodes 120-127 and 247 are the KAFIX instruction. See KA10 modifications for an explanation.

Opcode 257 is CONS. No one uses the CONS instruction anymore. If you want to know how it works, read the prints. Bit 18 in CONI APR, is the CONS flag, which interrupts on the APR channel. Bit 20 in CONO APR, clears the CONS flag.

The PDP-6 has been modified to have address condition switches like those on the KA10. These switches are located in 166 bay 1.

The SKIPx instructions on the PDP-6 have been fixed to avoid unnecessary memory store cycles.

The PDP-6 has been fixed so that user mode LUUOs, opcodes 001-037, will trap to User 40 and 41, instead of to Exec 40 and 41.

The PDP-6 has been modified so that an IBP instruction with a non-zero AC field will load the AC with the (incremented) byte pointer.

CONI TTY, bit 27 = 1 (for processor identification)

KA10 modifications:

Opcode 247 is KAFIX. This instruction does a floating point unnormalized add immediate and does not store an exponent. Commonly, it is used to fix and scale floating point numbers. KAFIX AC,233000 will fix the floating number in AC with result to AC. Other values of the effective address can produce scaling by powers of 2.

The KA10 Address Condition logic has been modified by the addition of two mode switches (located in bay 2). These are denoted OLD/NEW and USER/EXEC. In OLD mode, the USER/EXEC switch is ignored. The AS=RLA condition is generated from AS=MA. In NEW mode, the AS=RLA condition requires both AS=MA and the state of user mode addressing (i.e., EX REL A) must correspond to the USER/EXEC switch.

Bit 22 in CONO PI, sets CPA MEM PROT flag.

CONI PTP, bit 28 = KEY NXM STOP

CONI TTY, bit 28 = 1 (for processor identification)

PUSHJ doesn't clear the BIS flag.

JRST 10, and JRST 4, are illegal in User IOT mode

KA10 Modifications Resulting from the Installation of the BBN Pager

The A.I. Lab KA10 is equipped with a BBN Pager. All the appropriate modifications that go with the pager have been done. These include additional instructions, the XCT modifications, the inter-processor interrupt feature, etc. A brief description of these features appears below. For greater detail, refer to the BBN Pager documentation manual.

There are three new switches in bay 1 of the KA-10 pertaining to the BBN Pager:

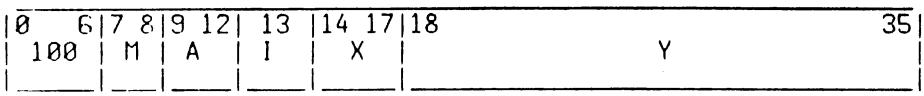
PGR DIS disables the the pager from seeing the signals EX REL A and MAPAC.

MAPAC DIS disables the pager from seeing MAPAC. This disables the AC base register feature of the pager.

JSYS DIS causes all JSYS instructions to execute as User JSYS instructions.

UMOVEx

The UMOVEx class of instruction modifications forces MOVEs to and from User space.

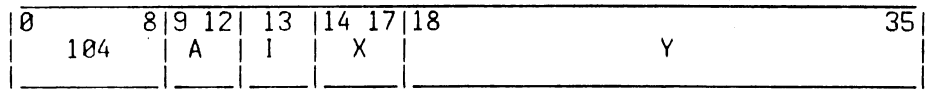


Move one word from the source to the destination specified by M, using the protection/relocation registers or the user address map if mapping is enabled. The source is unaffected, the original contents of the destination are lost.

UMOVE	User move	100
UMOVEI	User move immediate	101
UMOVEM	User move to memory	102
UMOVES	User move to self	103

These instructions provide a convenient way for the monitor to invoke the User address mapping to fetch or store information into the User address space (UMOVE or UMOVEM). UMOVEI provides a way for the monitor to do address computation using indirect addressing through the User address space. Of course, indexing and AC references are not affected by the choice of User map/EXEC map. However, addresses which indirect through the User ACs are handled specially (see section on Call From Monitor Flag).

JSYS Jump to System



If the effective address is 1000 or greater this is a User JSYS (at Stanford A.I. Lab all JSYS instructions are User JSYS instructions). Fetch a double (left half, right half) pointer from the effective address. Store the flags and incremented PC in the location indicated by the left half pointer and jump to the location indicated by the right half pointer. The flags and PC are stored in the left and right halves respectively of the indicated word.

If the effective address is less than 1000, the instruction is an Exec JSYS (at Stanford A.I. Lab there

are no Exec JSYS instructions). Enter monitor mode (leave user mode) and fetch a double pointer from the real core address (unmapped) equal to 1000 + the effective address. As above, the flags and PC are stored through the left half pointer and control jumps (in the monitor mode) to the location indicated by the right half pointer. If this instruction is executed in Exec mode, bit 7 of the PC flag word is affected.

XCT AC,E (where AC is non-zero)

XCT AC,E in Exec mode causes the instruction in location E to be executed with the memory reference(s) relocated by the current contents of the protection/relocation register or by the map if mapping is enabled. The different parts of the instruction referenced are relocated or not depending on the AC field of the XCT. The AC field bits have the following function:

10	4	2	1
Efective address computation, Last address of BLT	Data fetch, Byte pointer fetch, Pop stack, Push memory, 'From' of BLT	Address computation from byte pointer	Data store, Byte fetch/store, Push stack, Pop memory, 'To' of BLT

The 10 bit maps only the memory references of the efective address calculation. The last address transferred to by a BLT is mapped when this is on.

The 4 bit maps any data fetch, the fetch of a byte pointer, the location of the stack on a POP, the memory location of the data on a PUSH, and the 'from' address on a BLT.

When the 2 bit is on, any memory fetches during the address calculation of the address of a byte are mapped.

The 1 bit maps any data store, the fetch or store of a byte, the location of the stack on a PUSH, the memory location of the data store on a POP, and the 'to' address on a BLT.

Note that during address calculations, only memory fetches getting data for the computation are mapped.

Bit 7 of the KA10 PC flags word is used to store the state of a flip flop named CALL FROM MONITOR. This bit is saved and restored in the same fashion as the other PC flag bits. It is cleared by MR START, and set whenever an Exec JSYS is executed in Exec mode. This bit indicates to the called JSYS routine that effective addresses, byte pointers, and BLT pointers passed as arguments should refer to the Exec mode address space, not the current User address space. When this bit is on, special XCT and UMOVEx references are automatically forced into the Exec mode address space instead of the User's space.

This feature simplifies the coding of Exec JSYS routines which accept pointers as arguments and which may be called either from User mode or Exec mode. The routine merely makes use of any pointers with UMOVE, or special XCT instructions, and the CALL FM MON flag automatically forces references into the correct address space.

Device OAP (Other Arithmetic Processor, Device code = 20)

CONO OAP, bit 28 clear IPI APR flag
bit 29 set other processor's IPI APR flag
CONI OAP, bit 28 IPI APR flag
bit 29 Other processor's IPI APR flag
The IPI APR flag being set requests an interrupt on the APR channel.

Store Cycle Differences between the PDP-6 and KA10

The PDP-6 stores the results of any instruction in the order: memory, AC1, AC2. PDP-10s store the results in the order: AC1, memory, AC2. Two instructions behave slightly differently as a result. Good programming practice avoids both of these situations.

First, the instruction POP AC,AC on the PDP-6 will store the decremented stack pointer in the AC. In PDP-10s the word from the stack will be stored in AC

The second case is a BLT instruction in which the BLT AC is within the destination range of the BLT other than at the end of the destination range. In this instruction if an interrupt occurs just when the memory word being transferred is about to overwrite the BLT AC then in the PDP-6 the pointer word is written over the memory word. In the KA10 the memory word is written over the pointer word. Good programming practice is to avoid clobbering the BLT AC, except in the last word of a transfer.

Miscellaneous Differences Between the KA10 and the PDP-6:

When the READ IN switch is actuated on the PDP-6, the effect is the same as pushing start, except that the machine will start executing in shadow memory (real core locations 0-17, not the AC's). The PDP-10 however sends a pulse on the I-O buss to a specific device to load core.

The PDP-6 turns off the byte increment suppression flag when the PC and flags are being saved at a UUO or PI trap. The PDP-10 turns it off whenever the PC and flags are saved.

The MI cannot be loaded by a program on the PDP-6.

JRST 2,A(XR) may be used to get an indexed return along with flag restore from the index register. However on the PDP-6 the flags are restored from the left half of the full-word sum of the contents of XR and the JRST instruction itself. This is not usually the desired effect, and indirect addressing should be used instead. On the PDP-10 this instruction restores the flags from the left side of XR which is considered the correct operation.

KL10, KA10, and 166 I/O Status Bits

A concise summary of I/O status bits for three processors appears below. Note that this describes the KA10 and 166 processors at the Stanford A.I. Lab which differ from standard processors.

CONO PI,

	166	KA	KL
18	unused	Clr Power Fail	Write even address parity
19	unused	Clr Parity Error	Write even data parity
20	Clr Parity Error	Dis Parity Error Int	Write even directory parity
21	Dis Parity Error Int	Enb Parity Error Int	unused
22	Enb Parity Error Int	Set CPA MEM PROT	Drop prog requests on sel chan
23	Clear PI system		
24	Request int on sel chn		[Interrupts requested by
25	Turn on sel chn		this function (Bit 24) can
26	Turn off sel chn		be dropped only by use of
27	Turn off PI system		bit 22.]
28	Turn on PI system		
29	Select channel 1		
30	Select channel 2		
31	Select channel 3		
32	Select channel 4		
33	Select channel 5		
34	Select channel 6		
35	Select channel 7		

CONI PI,

	166	KA	KL
0	unused	unused	unused
...			
10	unused	unused	unused
11	unused	unused	Prog request on channel 1
12	unused	unused	Prog request on channel 2
13	unused	unused	Prog request on channel 3
14	unused	unused	Prog request on channel 4
15	unused	unused	Prog request on channel 5
16	unused	unused	Prog request on channel 6
17	unused	unused	Prog request on channel 7
18	unused	Power fail	Write even address parity
19	unused	Parity error	Write even data parity
20	Parity error	Parity Int Enb	Write even directory parity
21	In progress on chn 1		
22	In progress on chn 2		
23	In progress on chn 3		
24	In progress on chn 4		
25	In progress on chn 5		
26	In progress on chn 6		
27	In progress on chn 7		
28	PI system on		
29	Channel active 1		
30	Channel active 2		
31	Channel active 3		
32	Channel active 4		
33	Channel active 5		
34	Channel active 6		
35	Channel active 7		

CONO APR,

	166	KA	KL
18	Clr PDLOV	unused	unused
19	Clr IOT user	IOB reset	IOB reset
20	Clr CONS flg	unused	Enb flgs selected by b24:31
21	unused	Clr Addr Brk flg	Dis flgs selected by b24:31
22	Clr MPV	Clr MPV	Clr flgs selected by b24:31
23	Clr NXM	Clr NXM	Set flgs selected by b24:31
24	Dis CLK Int	Dis CLK Int	SBUS error
25	Enb CLK Int	Enb CLK Int	NXM error
26	Clr CLK flg	Clr CLK flg	In-Out Page Fail
27	Dis PC chg Int	Dis FOV Int	MB parity error
28	Enb PC chg Int	Enb FOV Int	Cache directory error
29	Clr PC chg flg	Clr FOV flg	Cache address parity error
30	Dis AROV Int	Dis AROV Int	Power fail
31	Enb AROV Int	Enb AROV Int	Sweep done
32	Clr AROV flg	Clr AROV flg	unused
33	APR PIA 4		
34	APR PIA 2		
35	APR PIA 1		

CONI APR,

	166	KA	KL
0	unused	unused	unused
...			
5	unused	unused	unused
6	unused	unused	SBUS error enb
7	unused	unused	NXM error enb
8	unused	unused	In-Out Page fail enb
9	unused	unused	MB parity error enb
10	unused	unused	Cache directory error enb
11	unused	unused	Cache addr parity error enb
12	unused	unused	Power Fail enb
13	unused	unused	Sweep Done enb
14	unused	unused	unused
...			
18	CONS flg	unused	unused
19	PDLOV	PDLOV	Sweep Busy
20	User IOT mode	User IOT mode	unused
21	User Mode	Address Break	unused
22	MPV flg	MPV flg	unused
23	NXM flg	NXM flg	unused
24	unused	unused	SBUS error flg
25	CLK enb	CLK enb	NXM error flg
26	CLK flg	CLK flg	In-Out page fail flg
27	unused	¬CPA INT	MB parity error flg
28	PC CHG enb	FOV enb	Cache directory error flg
29	PC CHG flg	FOV flg	Cache addr parity error flg
30	unused	MA Trap offset	Power fail flg
31	AROV enb	AROV enb	Sweep done flg
32	AROV flg	AROV flg	Int requested flg
33	APR PIA 4		
34	APR PIA 2		
35	APR PIA 1		

PC and flags

	166	KA	KL
0	AR OV		AR OV/PCP [See note]
1	AR CRY0		
2	AR CRY1		
3	PC Change	FOV	
4	BIS		First Part Done
5	User		
6	User In-Out		User In-Out/PCU [See note]
7	unused	CAL FM MON	Public/Supervisor [See note]
8	unused	unused	Address Failure Inhibit
9	unused	unused	Trap 2
10	unused	unused	Trap 1
11	unused	FXU	
12	unused	DCK	
13-17	Zero		
18-35	PC		

[Note: In the KL10 some PC flags have different meanings in Exec mode than they have in User mode. For further details see the PC Flags section of Appendix L, KL10 Information.]

KL10 Information

The information that appears below applies mostly to systems programmers. Hopefully, it accurately reflects the state of the Stanford A. I. Lab KL10 (a revision 7A CPU). Some of the problems reported below may be fixed in the LOTS 2040 (revision 9 CPU) and in other late-model CPUs.

When taking an interrupt (e.g., by a JSR), some user mode PC flags are kept in the new (exec mode) PC. One of these is AR OV (bit 0) which regrettably means PCP (previous context public) in exec mode. Therefore, on all interrupts where user core will be referenced, PCP should be cleared in the exec mode PC.

CONO PI,<Turn off selected channel> is not effective immediately. This means that all channels' interrupt code must test for the condition of being in progress on a channel that has been turned off. If that condition is detected, and it isn't the case that this channel was GEN'ed on, then the interrupt should be dismissed.

CONO PI,<generate interrupt on selected channel> has two peculiarities. If the GEN is aimed at a higher priority channel then it isn't immediately effective. In this case, the CONO PI,<GEN channel> should be followed by a logical no-op that requires an EBUS cycle such as CONO PI,PION. If the GEN is aimed at a lower priority channel then one should be aware of the fact that the GEN isn't effective until after the present channel is dismissed. If it's necessary to get to a lower priority channel without returning to the interrupted process, then you must dismiss to an exec mode loop or something.

The KA10 long mode floating point instructions (FADL, FSBL, FMPL, FDVL) and two other floating point instructions, UFA and DFN, are not implemented in the 2040 microcode nor in the Stanford A.I. Lab KL10 microcode. These instructions will trap as UUOs and be simulated by the monitor, running 50 to 200 times slower than the KA10 versions. If a program uses these to implement KA10 style double precision floating point, then the program should be converted to use the KL10 hardware double precision format.

KL10 new instructions:

BLKO APR, Load cache refill algorithm RAM.
Writes into a 128x3 table. One BLKO per 3 bit entry.

```
Effective address of the instruction decodes as follows:
Bit
18      Data bit 4   New order bit
19      Data bit 2   New LRU 2
20      Data bit 1   New LRU 1
21:26   unused
27      RAM address 64   (From new MRU 2)
28      RAM address 32   (From new MRU 1)
29      RAM address 16   (From old MRU 2)
30      RAM address  8   (From old MRU 1)
31      RAM address  4   (From old order bit)
32      RAM address  2   (From old LRU 2)
33      RAM address  1   (From old LRU 1)
34:35   unused
```

Each (quadruple of) cache line(s) has a 5 bit field (Cache Use Bits) to specify which of the caches was LRU, MRU and the relative order of the other two caches. These five bits plus 2 bits of new MRU (i.e., the currently referenced cache) are used to index a 128 word table to produce 3 bits (2 bits of LRU and 1 bit of order). These 3 bits plus the 2 bits of new MRU, replace the 5 Cache Use bits.

Since the foregoing explanation is somewhat difficult to understand, the following table of 128

three-bit values is provided. These values are for the normal case of four caches enabled.

```
       00       10       20       30       40       50       60       70
000/   01234567 31232123 71271127 65675567 03230223 01234567 07770007 46664464
100/   31331113 07770007 01234567 45574547 01220121 05660560 45654564 01234567
```

BLKO PAG,E Invalidate page table entry corresponding to the (8) page(s) addressed by (E⊗-9)&770

DTE20 KL10 I/O Functions

CONO DTE0,

```
18:21          unused
22             DONG11  KL10 requests service by the 11
23             clear RELOAD PDP11 button
24             set RELOAD PDP11 button
25             unused
26             Clear DONG10    PDP-10 has responded to interrupt from 11
27             unused
28             unused
29             clear TO-11 Normal Termination and TO-11 Error Termination
30             clear TO-10 Normal Termination and TO-10 Error Termination.
31             enable loading bits 32:35
32             Set PI0 ENABLE (if enabled by bit 31)
                      [This is ignored on a privileged 11]
33:35          PIA
```

CONI DTE0, [Bits marked with * request interrupts]

```
00:19          unused
20             1 if this DTE is restricted, 0 if privileged.
21             PDP-11 Power failure
22             DONG11  KL10 requests service by the 11
23             unused
24             unused
25             unused
26             *DONG10 PDP-11 requests an interrupt on the KL10
27             *TO-11 Error Termination
28             unused
29             *TO-11 Normal Termination
30             *TO-10 Normal Termination
31             *TO-10 Error Termination
32             PI0 Enable
33:35          PIA 4,2,1 (1)
```

DATAI DTE0, unassigned. Makes EBUS parity errors.

DATAO DTE0,E

```
00:22          unused
23             bit → TO-10 I BIT
24:35          bits → EBHOLD 11-00
               [Bit 23 a zero means that when the TO-10 transfer completes,
               only the 10 (not the 11) will be interrupted.  This
               permits scatter write under control of the 10]
               [Bits 24:35 are a negative byte count of the number of bytes
               to transfer before giving a normal termination interrupt.
               0 means transfer 0 bytes.]
```

BLKO PI, SBUS diagnostic function.

The data at the effective address is written on the SBUS as a diagnostic function. The result of executing this function is read back from the SBUS and stored in the effective address plus one. At present two kinds of devices may be addressed by the SBUS diagnostic functions. These are the DMA20 external memory adapter, and the MA20 internal memory. Each device supports two diagnostic functions. The functions and results are set forth in the following table:

	MA20				DMA20			
	Write		Read		Write		Read	
Bit	Fcn 0	Fcn 1	Fcn 0	Fcn 1	Fcn 0	Fcn 1	Fcn 0	Fcn 1
00	0	0			0	0	0	
01	0	0			0	0	0	
02	0	0	Inc RQ		1	1	NXM	
03	x2	x2			0	0	RD Par	
04	x1	x1		SM 3	0	0	WR Par	
05	Clr Err		ADR Par	SM 2	Clr Err		ADR Par	
06	INTL 2		INTL 2	SM 1	INTL 2		INTL 2	
07	INTL 1		INTL 1	SM 0	INTL 1		INTL 1	
08	RQ EN 0			0			SB RQ 0 0	
09	RQ EN 1			0			SB RQ 1 0	
10	RQ EN 2			0			SB RQ 2 1	
11	RQ EN 3			1			SB RQ 3 0	
12	EN 6:11	Loop Ar		Loop Ar	EN 6:7	Loop Ar	RD RQ	Loop Ar
13							WR RQ	
14		Ad Sw 14		Ad Sw 14			SB AD 14	
15		Ad Sw 15		Ad Sw 15			SB AD 15	
16		Ad Sw 16		Ad Sw 16			SB AD 16	
17		Ad Sw 17		Ad Sw 17			SB AD 17	
18		Lo Ad 18		Lo Ad 18			SB AD 18	
19		Lo Ad 19		Lo Ad 19			SB AD 19	
20		Lo Ad 20		Lo Ad 20			SB AD 20	
21		Lo Ad 21		Lo Ad 21			SB AD 21	
22		Hi Ad 18		Hi Ad 18			SB AD 22	
23		Hi Ad 19		Hi Ad 19			SB AD 23	
24		Hi Ad 20		Hi Ad 20			SB AD 24	
25		Hi Ad 21		Hi Ad 21			SB AD 25	
26		EN 14:25					SB AD 26	
27		Cur Marg					SB AD 27	
28		Stb Marg					SB AD 28	
29		VTH Marg					SB AD 29	
30		Mar Dir		Marg Sel			SB AD 30	
31					0	0	SB AD 31	
32					0	0	SB AD 32	
33					0	0	SB AD 33	
34					0	0	SB AD 34	
35	0	1			0	1	SB AD 35	

INTL 2	1		RQ EN 0	1	2	3	
0	0	Offline	0	0	0	0	Controller offline
0	1	MA20: 1 way interleaved; DMA20: 1 bus mode	1	0	1	0	Even Controller; 2 or 4 way interleaved
1	0	MA20: 2 way interleaved; DMA20: 2 bus mode	0	1	0	1	Odd Controller; 2 or 4 way interleaved
1	1	MA20: 4 way interleaved; DMA20: 4 bus mode	1	1	1	1	Odd and Even Controller; no interleaving

In the DMA20 bits 31:35 are the function select code. In the MA20 bit 35 alone selects the function. In function 1, bits 8:11 read back the controller type; 1 = MA20, 2 = DMA20. In the MA20, Mar Dir = 0 means low margins; Mar Dir = 1 means high margins. However, Mar Dir = 1 and bits 28:30 all zero means clear margins. The bits x1 and x2 are the MA20 device address; up to four MA20 controllers may be present on the SBUS. A bit name of the form "EN a:b" means that a one in this bit enables bits a:b to be loaded in this operation. Loop Ar means "Loop Around". Loop Around mode is for data path debugging: the DMA20 will not cycle the buses nor will the MA20 actually sense the memories.

KL10 PC Flags

PCU

In exec mode, bit 6 of the PC flags is interpreted as PCU (Previous Context User). It is this bit that causes PXCT to attempt to reference the target address in user mode. In user mode the bit is IOT USER. It can be set in any of the following ways:

1. By a PC storing interrupt instruction if interrupting out of user mode.

2. By a flag loading instruction (JRSTF, MUUO, etc) if specified by the data (i.e. bit 6 of the word the flags are loading from) and if the machine is either not in user mode or is leaving user mode.

It can be cleared in two ways:

1. By attempting to load it with zero (no restrictions).

2. A condition that would leave user mode will clear PCU if some other condition doesn't try to set it at the same time. For example, interrupting out of exec mode will clear PCU.

Therefore, PCU will be set automatically by interrupting out of user mode (if the instruction in the interrupt location is a PC storing instruction) and upon MUUO execution if the UPT location specifying the new PC has bit 6 on.

PUBLIC

The PUBLIC bit is bit 7 of the PC word and divides user mode and exec mode into two submodes. In exec mode, PUBLIC off is KERNEL mode, PUBLIC on is SUPERVISOR mode. In user mode, PUBLIC off is CONCEALED mode, PUBLIC on is PUBLIC mode. Restrictions on which instructions are legal are determined by exactly which of these four modes the machine is in. Switching between the various modes is controlled by the PUBLIC and USER mode PC bits and by the page table PUBLIC bit. These bits may sometimes be set and cleared by a JRST 2, and can always be set and cleared by an MUUO new PC word. In KERNEL mode everything is legal, there are no restrictions. However, if an instruction is fetched out of a PUBLIC page (one whose page table PUBLIC bit is set), the PUBLIC bit is set in the PC flags and the machine enters SUPERVISOR mode. SUPERVISOR mode may be left in two ways. First, by entering USER mode with a JRST 2, (or MUUO for that matter). In this case the PUBLIC bit is loaded without restriction from the new PC word. Second, by entering KERNEL mode by transferring to a PORTAL instruction (a JRST 1,) in a private page (one whose page table PUBLIC bit is off). An analogous situation occurs in USER mode. A program running in CONCEALED mode that fetches an instruction from a PUBLIC page will enter PUBLIC mode; a PUBLIC mode program may enter CONCEALED mode by transferring to a PORTAL instruction in a private page.

SUPERVISOR mode is not exactly analogous to USER mode in that SUPERVISOR mode may read from a private page but not write, whereas USER mode may not read or write a private page. Finally, both SUPERVISOR and USER modes may not transfer control to a private page except at a PORTAL (JRST 1,) instruction.

PCP

In exec mode, bit 0 of the PC represents SCD PCP (except in JFCL 10, which tests AROV). SCD PCP is loaded from bit 0 of a PC word restored by JRST 2, (or from a UUO or trap new PC word). PCP means previous context public, i.e., supervisor was called from a public, not a concealed,

program.

When a PXCT instruction occurs, the target instruction will be executed in PUBLIC mode if PCP is set. Note that PCP is not affected by interrupting, so that a user mode program may set PCP with a JRST 2, (or by making an arithmetic overflow). When the machine enters exec mode for an interrupt, PCP will still be set; it is the responsibility of the system code to make sure PCP has the right value before the first PXCT is done.

PXCT - Previous Context Execute (XCT in Exec mode with a non-zero AC field)

The AC field of the PXCT is decoded to determine when to enforce previous context. Note that previous context may be exec mode as well as user mode. The previous context ACs are determined by DATAO PAG, bits 9-11. For previous context non-ac references, the PCU bit in the PC word (bit 6) forces user mode previous context references. If a target instruction of PXCT does anything at all unusual with its memory references, then one ought to read the microcode for that instruction before assuming that PXCT will work reasonably. The PXCT AC field bits control which parts of the target instruction are interpreted in previous context mode as follows:

AC field value — part of cycle done in previous context

10 Effective address computation of target instruction. i.e., for indexing, the previous context ACs are used. If the effective address computation of the target instruction involves an indirect cycle, then the location that is read will be in the previous context if both this bit and the 4 bit are on (the 4 bit enables reads and writes).

4 Any memory reference, whether reading or writing, that is specified by E. i.e. the fetch or store of the effective address, the source of a PUSH, or the destination of a POP or a BLT.

2 Effective address computation of a byte pointer. i.e., indexing specified by a byte pointer.

1 Stack word in PUSH or POP, the source in a BLT, the fetch of the effective address of the byte pointer in a load byte instruction.

Clocks and meters

There are 5 "clocks", each is actually a counter of some kind:

Interval	100 KHz	12 bits EPT 514 (vector)
Time base	1 MHz rate	52 bits EPT 510-511
Performance		52 bits EPT 512-513
EBOX cycle count	1/2 EBOX clock rate	52 bits UPT 504-505
MBOX cycle count	1 per EBOX mem req.	52 bits UPT 506-507

The MBOX clock counter and EBOX clock counter are together referred to as the "Accounting Meters." All 52 bit clocks actually are double words in memory with 16 bits of each counter implemented in hardware too. The 52 bit clocks are read with DATAI or BLKI which produce a double word result suitable for DADD etc. The low 12 bits of the low order counter words are zero (in the hopes that a faster, compatible format clock will be available on future machines).

MTR is 24, TIM is 20.

CONO MTR, Accounting and Timebase Control, Interval Timer PIA

18 Enable bits 21:23

19	unused
20	unused
21	PI Acct Enable
22	Exec Acct Enable
23	Acct. On
24	Turn off timebase
25	Turn on timebase
26	Clear Timebase
27	unused
28	unused
29	unused
30	unused
31	unused
32	unused
33	Interval timer PIA 4
34	Interval timer PIA 2
35	Interval timer PIA 1

The accounting meters are enabled by:

(Acct On) ∧
(User ∨ ¬PI in progress ∨ PI Acct Enable) ∧
(User ∨ PI in prog ∨ Exec Acct Enb)

When turned on, the accounting meters are enabled to count in user mode, regardless of PI level. Also, if Exec Acct Enb is true, meters are enabled at UUO level (exec mode not PI in progress). Also, if PI Acct Enable is true, meters are enabled while PI in progress. The meters are disabled by either PI cycle or Ucode State 3.

When enabled, the MBOX meter counts when the MBOX finishes a memory cycle requested by the EBOX. When enabled, the EBOX meter counts every second MBOX clock tick unless the EBOX is waiting for the MBOX.

CONI MTR,

0:17	Unused
18	unused
19	unused
20	unused
21	PI Acct Enable
22	Exec Acct Enable
23	Acct on
24	unused
25	Timebase on
26	unused
27	unused
28	unused
29	unused
30	unused
31	unused
32	unused
33	Interval meter PIA 4
34	Interval meter PIA 2
35	Interval meter PIA 1

CONO TIM,

18	Clear Up Counter
19	unused
20	unused
21	1=Turn on, 0=turn off interval timer
22	Clear Interval Done and Interval Overflow
23	Unused
24:35	Load Period Register

CONI TIM,

0:5	Unused
6:17	Up counter
18:20	unused

```
21              Interval Timer On
22              ±Interval Timer Done
23              ±Interval Overflow
24:35           Period register
                (When the Up Counter reaches the value specified in
                the period register, it sets Interval Timer Done
                and resets the up counter to zero.
                Interval timer interrupts execute location 514 in the EPT.
                If you set a period that's smaller than the current
                value of the Up Counter,  Up Counter Overflow will
                be set (after the up counter reaches maximum value).)
```

Timebase:

DATAI TIM, Read doubleword timebase.

Accounting meters:

DATAI MTR, Read doubleword EBOX meter.
BLKI MTR, Read doubleword MBOX meter.

DATAO PAG, with bit 2 set will reset the UBR. Unless bit 18 is set to inhibit storing accounting meters (useful if there's no old UPT), the old EBOX and MBOX counters are added to the old UPT 504-507. The meters are cleared.

Performance Analysis

BLKO TIM, Set performance counter enables.

```
 0              Internal Channel 0 Busy Enb                 A
 1              Internal Channel 1 Busy Enb                 A
 2              Internal Channel 2 Busy Enb                 A
 3              Internal Channel 3 Busy Enb                 A
 4              Internal Channel 4 Busy Enb                 A
 5              Internal Channel 5 Busy Enb                 A
 6              Internal Channel 6 Busy Enb                 A
 7              Internal Channel 7 Busy Enb                 A
 8              Internal Channel Don't Care                 A
 9              Microcode State Don't Care                  B
10              ECL probe low PA enb                        C
11              ECL probe don't care                        C
12              ¬Cache Request Enb                          D
13              ¬Cache Fill (miss) Enb                      D
14              ¬Cache (Ebox) writeback Enb                 D
15              ¬Cache Sweep Enb                            D
16              Cache don't care                            D
17              unused
18              PI 0 PA enb                                 E
19              PI 1 PA enb                                 E
20              PI 2 PA enb                                 E
21              PI 3 PA enb                                 E
22              PI 4 PA enb                                 E
23              PI 5 PA enb                                 E
24              PI 6 PA enb                                 E
25              PI 7 PA enb                                 E
26              No PI PA enb                                E
27              User PA enb                                 F
28              Mode PA don't care                          F
29              1=event mode; 0=Duration mode
30              clear perf meter
31              Unused
32              Unused
33              unused
34              unused
35              unused
```

BLKI TIM, Read doubleword performance analysis counter.

A Boolean condition consisting of the AND of six terms is formed. In duration mode, while this condition is true, the performance meter is counted at half the MBOX clock rate. In event mode, every transition of this condition from False to True is counted. The condition is the AND of six terms (designated A, B, C, D, E and F, to correspond to the description of BLKO TIM, above).

```
A:      ((Internal Channel n Busy)∧(Internal channel n Busy Enb)) ∨
              (Internal channel don't care)

B:      (Microcode state 01) ∨ (Microcode state don't care)

C:      ((ECL probe state low enb)≠(Probe)) ∨ (ECL Probe don't care)

D:      ((Fill cache read) ∧ (Cache fill enb)) ∨
        ((EBOX waiting) ∧ (cache request enb)) ∨
        ((EBOX writeback) ∧ (EBOX writeback enb)) ∨
        ((Sweep writeback) ∧ (sweep writeback enb)) ∨
        (Cache don't care)

E:      ((PI n is highest channel held or cycling) ∧ (PI n PA enb))  ∨
        ((No PI channel is held)∧(No PI PA enb))

F:      ((User mode)≡(User PA enb)) ∨ (Mode PA don't care)
```

Supplement To DECsystem-10 Hardware Reference Manual Extended Instruction Set

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

1st Edition, March 1976

Copyright © 1976 by Digital Equipment Corporation

The material in this manual is for informational purposes and is subject to change without notice.

Digital Equipment Corporation assumes no responsibility for any errors which may appear in this manual.

Printed in U.S.A.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

FLIP CHIP	DECtape
UNIBUS	DECsystem-10
PDP	DECmagtape
DECUS	DIGITAL
DEC	

TABLE OF CONTENTS

TABLE OF CONTENTS (Cont)

ILLUSTRATIONS

TABLES

EXTENDED INSTRUCTION SET

INTRODUCTION

The Extended Instruction Set is a multi-functioned instruction which performs character string editing, decimal to binary conversion, binary to decimal conversion, string move with left or right justification, string move with offset or translation, and string compare.

The Extended Instruction Set consists of a single KL10 instruction (for which the mnemonic is "EXTEND", and the op-code is 123), and a set of 15 Extended operators, which themselves have the standard KL10 instruction format.

The Extended operator word is addressed by the "Extend" instructions effective address and is evoked as though by an XCT. It is the op-code contained in bits 0-8 of the Extended operator word which defines the type of operation to be performed.

The set of 15 operators are functionally divided into four groups. These groups, together with the discrete Extended op-codes, mnemonics and meaning are listed below.

NOTE

Digital Equipment Corporation reserves the right to change any features of the extended instruction set that are not explicitly documented in this supplement. Fields of word formats designated "not used" are reserved for future use by Digital Equipment Corporation, and should be left as 0.

GROUP	EXTENDED OP-CODE	MNEMONIC	MEANING
COMPARE	001	CMPSL	Compare strings, skip if less.
	002	CMPSE	Compare strings, skip if equal.
	003	CMPSLE	Compare strings, skip if less or equal.
	005	CMPSGE	Compare strings, skip if greater or equal.
	006	CMPSN	Compare strings, skip if not equal.
	007	CMPSG	Compare strings, skip if greater.
EDIT	004	EDIT	Process string according to pattern.
CONVERT	010	CVTDBO	Convert decimal to binary by offset.

GROUP	EXTENDED OP-CODE	MNEMONIC	MEANING
CONVERT (Cont.)	011	CVTDBT	Convert decimal to binary by translation.
	012	CVTDBO	Convert binary to decimal with offset.
	013	CVTBDT	Convert Binary to decimal with translation.
MOVE	014	MOVSO	Move string with byte offset.
	015	MOVST	Move string with byte translation.
	016	MOVSLJ	Move string unmodified with left justification.
	017	MOVSRJ	Move string unmodified with right justification.

The Programmer must set up a number of Parameters prior to performing the "Extend" instruction. This includes one of the blocks of AC's of the type shown in Figure 1. The address of the block of

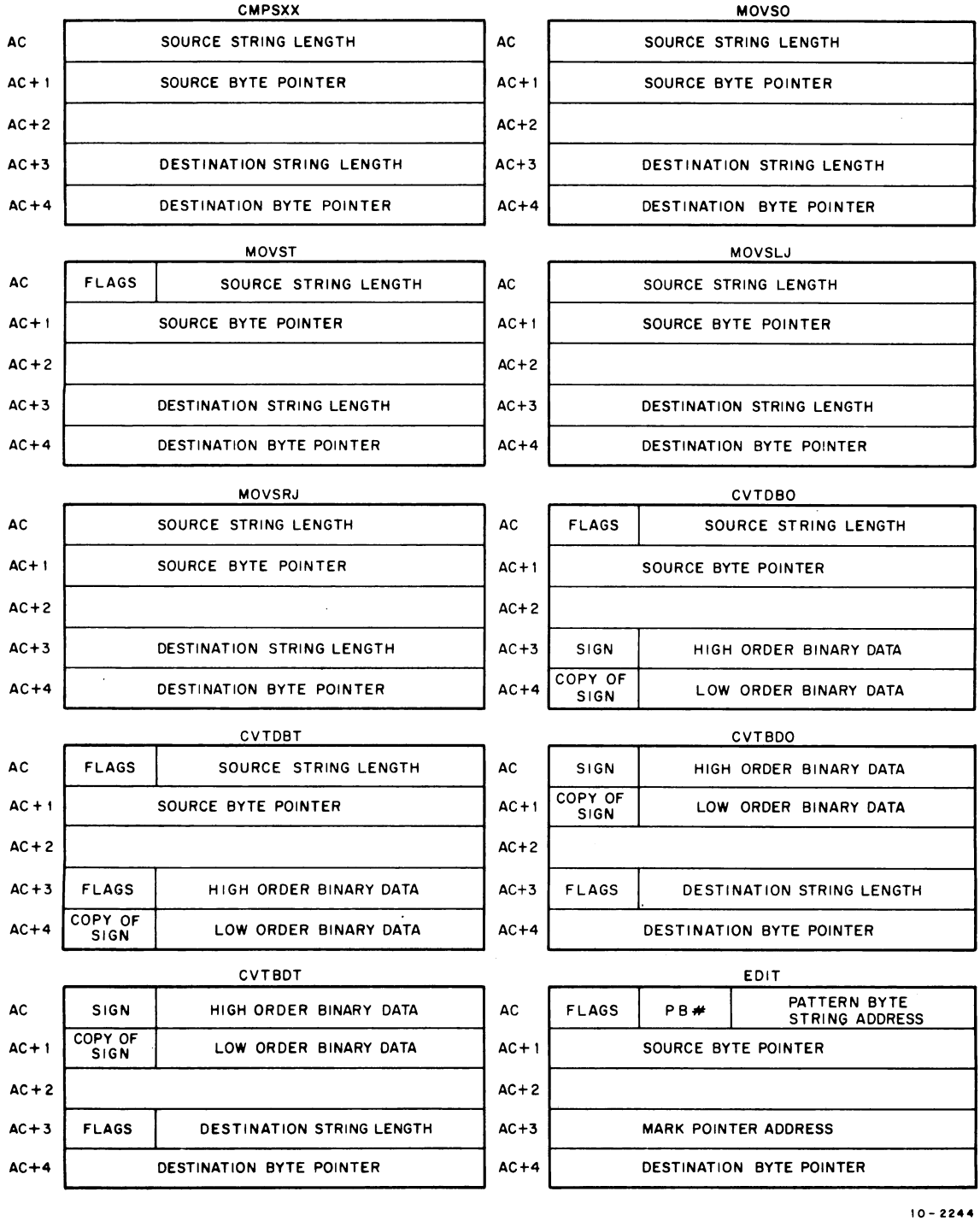


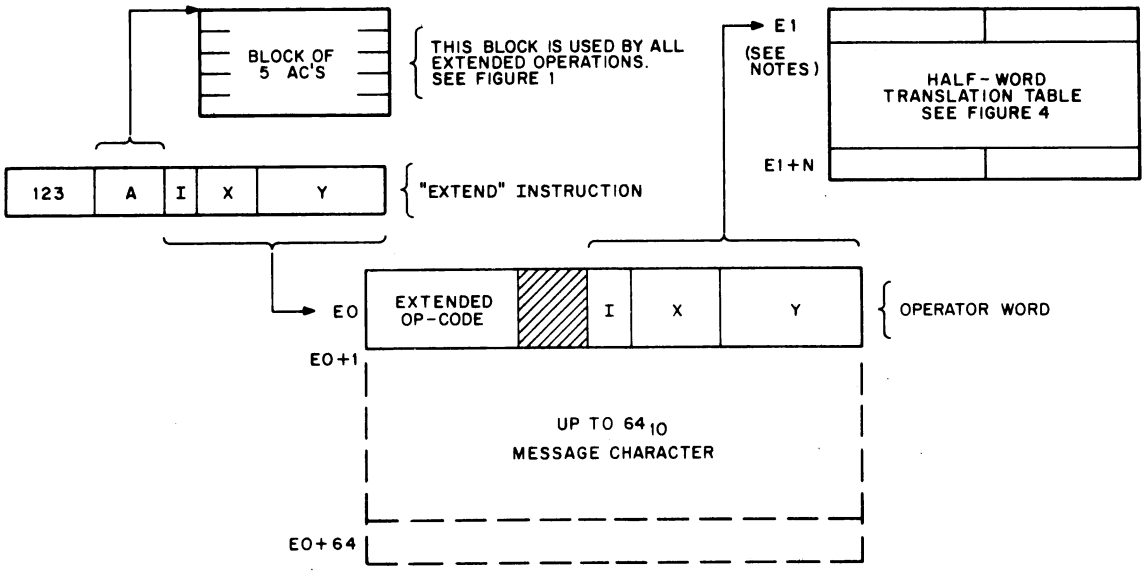
Figure 1 Accumulator Block Types

AC's is provided by the A field (bits 9-12), of the "Extend" instruction. Referring to Figure 1, notice that the interpretation of the contents of the block depends on the type of Extended operator being performed.

Additional parameters provided include a byte offset value which is used by Extended operators which include "Offset" in their names, or the address of a halfword translation table which is used by EDIT, and also by Extended operators which include "TRANSLATION" in their names.

EDIT requires special characters to be inserted into a destination string. These characters are referred to as message characters. Up to 64_{10} such characters may be defined. The first of these is called the fill character, and is the word at location E0+1. Several of the operators utilize this word to "Fill" a string to meet some length constraint. Figure 2 illustrates the basic linkage between the "EXTEND" instruction, the operator word and associated parameter storage.

To execute the operator (XCT) means that the processor performs the operator as though it were an instruction and performs it out of the normal sequence, i.e., the sequence defined by the program counter. Thus in this case, the operator is fetched not from an address supplied by PC, but from an address supplied by the "EXTEND". instruction.



NOTES:

1. For MOVSO, CVTDBO, and CVTBDO, E1 is a Fixed Byte Offset.
2. For EDIT, MOVST, CVTDBT, and CVTBDT, E1 is the Base Address of the Translation Table.

10-2245

Figure 2 Linkages

Control will be returned to PC at either the location one greater than the address containing the "EXTEND" instruction (abnormal or non-skip return), or the location two greater than the address containing the "EXTEND" instruction (normal or skip return).

Table 1 lists each of the operators together with an indication of the reason for the type or return address provided by the "EXTEND" instruction.

TABLE 1

EXTENDED OPERATORS

EXTENDED OPERATOR	PC ADDRESS	REASON FOR RETURN
CMPSXX	PC+1	Skip condition not satisfied.
CMPSXX	PC+2	Skip condition satisfied.
MOVSO	PC+1	The size of the byte with offset added is inconsistent with the destination byte pointer.
MOVSO	PC+2	The entire source string, (offset one byte at a time), was inserted in the destination string.

TABLE 1 (Cont.)

EXTENDED OPERATORS

EXTENDED OPERATOR	PC ADDRESS	REASON FOR RETURN
MOVST	PC+1	A halfword entry in the translation table accessed by the MOVST operator contained the "ABORT CODE".
MOVST	PC+2	The entire source string was translated.
MOVSLJ	PC+1	The destination string is shorter than the source string. The source string is truncated to fit. The source byte pointer is left at the point of truncation.
	PC+2	This operator skips when the destination string is equal to or longer than the source string.

TABLE 1 (Cont.)

EXTENDED OPERATORS

EXTENDED OPERATOR	PC ADDRESS	REASON FOR RETURN
MOVSRJ	PC+2	This operator always skips. NOTE The word immediately following the MOVSRJ instruction must be a no-op instruction for compatibility with possible future variations in what MOVSRJ will do. The source count and byte pointer are undefined and should not be referenced.
CVTDBO	PC+1	A halfword entry in the translation table was accessed by the CVTDBO operator and the byte offset was added to this. The result was not a digit between 0 and 9.
CVTDBO	PC+2	The entire source string was converted.

TABLE 1 (Cont.)

EXTENDED OPERATORS

EXTENDED OPERATOR	PC ADDRESS	REASON FOR RETURN
CVTDBT	PC+1	A halfword entry in the translation table was accessed by the CVTDBT operator, and the translation function was not a digit between 0 and 9, or the translation function contained the "ABORT CODE".
CVTDBT	PC+2	The entire source string was converted.
CVTBDO	PC+1	The length count specifying the number of digits to be stored in the destination string is less than the number of digits that will actually be generated if the conversion is performed.
CVTBDO	PC+2	The entire double-word binary integer was converted by offset.

TABLE 1 (Cont.)

EXTENDED OPERATORS

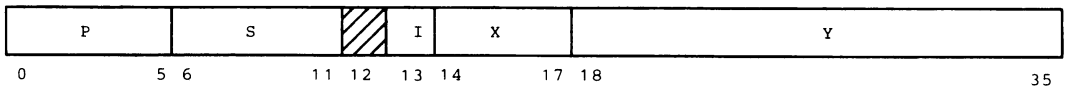
EXTENDED OPERATOR	PC ADDRESS	REASON FOR RETURN
CVTBDT	PC+1	See CVTBDO.
CVTBDT	PC+2	The entire double-word binary integer was converted by translation.
EDIT	PC+1	A halfword entry in the translation table accessed by the EDIT operator contained the "ABORT CODE".
EDIT	PC+2	The EDIT operation successfully performed a stop pattern byte operator which is the normal way to Terminate Editing.

STRING REPRESENTATION

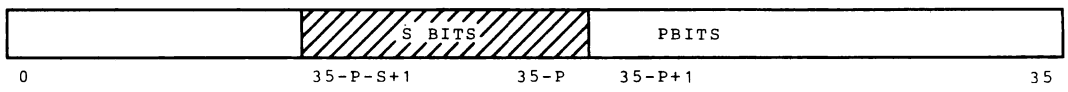
A string consists of a sequence of one or more equal size bytes. Bytes of the string are always tightly packed in a word, but the string need not begin nor end on a word boundary and in some cases,

bytes of adjacent words are not physically contiguous, e.g., there is a residue of unused bits between words.

Strings are addressed utilizing standard KL10 byte pointers of the form:



where S is the size of the byte as a number of bits, and P is its position as the number of bits remaining at the right of the byte in the word. The rest of the pointer is interpreted in the same way as an instruction: I, X, and Y are used to calculate the address of the location that is the source or destination of the byte. Thus, the pointer aims at a word whose format is:



where the shaded area is the byte. The byte pointer is always incremented prior to obtaining the byte, i.e., modified so its points to the next byte position in a set of memory locations. Bytes are processed from left to right in a word so incrementing merely replaces the current value of P by P-S, unless there is in-

sufficient space in the present location for another byte of the specified size (P-S < 0). In this case, Y is increased by one to point to the next consecutive location, and P is set to 36-S to point to the first byte at the left in the new location.

DOUBLE-PRECISION BINARY INTEGER

The binary integers accepted as input by binary to decimal conversion (CVTBDO, CVTBDT), and produced by decimal to binary conversion (CVTDBO, CVTDBT) have the same format as double-word integers produced and accepted by other instructions in the KL10, e.g., MUL, DIV, and the double-word integer arithmetic instructions DADD, DSUB, DMUL, and DDIV. This format is illustrated in Figure 3.

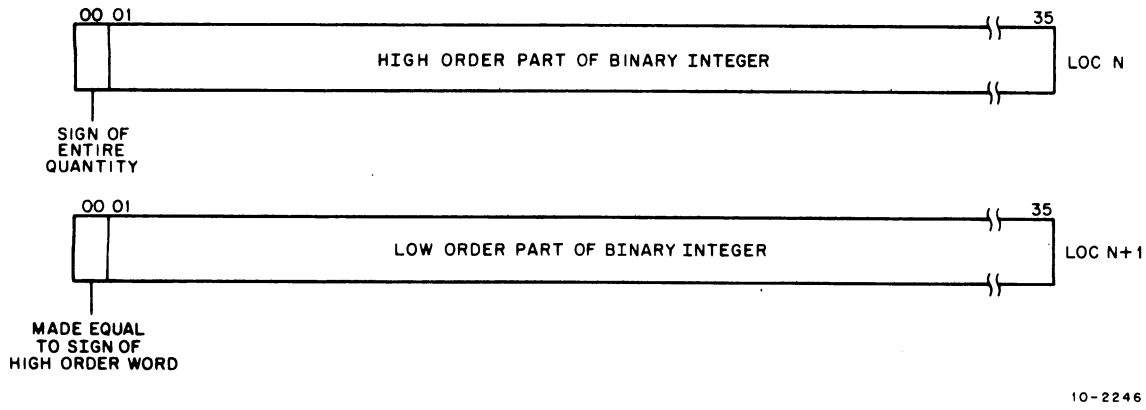


Figure 3 Double-Precision Binary Integer Format

As indicated in Figure 3, integers are represented in two's complement in two adjacent words. The most significant part is in the word with lowest address (N), and that word carries (in bit 0) the

sign of the entire quantity. The least significant part occupies bits 1-35 of the word with highest address (N+1). Bit 0 of that word is ignored in all operands, and is made equal to the sign of the answer in all results.

OFFSETTING

The OFFset function consists of adding E1 (the byte OFFset), to specified bytes taken from a source. In move string offset MOVSO, and convert decimal to binary CVTDBO, the source is a byte string addressed by a standard KL10 byte pointer.

In convert binary to decimal offset CVTBDO, the source is a double word binary integer and the bytes, which are to be offset, are individual decimal digits formed from the magnitude of the double-word binary integer.

TRANSLATION

Translation consists of conditionally replacing bytes in a destination string addressed by a standard KL10 byte pointer, with a preset representation of that byte, obtained from a table at location E1.

The Translation function consists of a selected halfword composed of the byte representation (up to 15 bits are provided) and except in the case of convert binary to decimal translated CVTBDT, a three

bit code which provides the disposition of the translation function, e.g., there is no byte representation for the current byte being translated so do not store anything in the destination string for the current byte or abort the entire operation and so forth.

In the case of CVTBDT, the entire halfword is the digit representation for the converted binary source byte. These tables are covered in greater detail in the sections describing the individual operators.

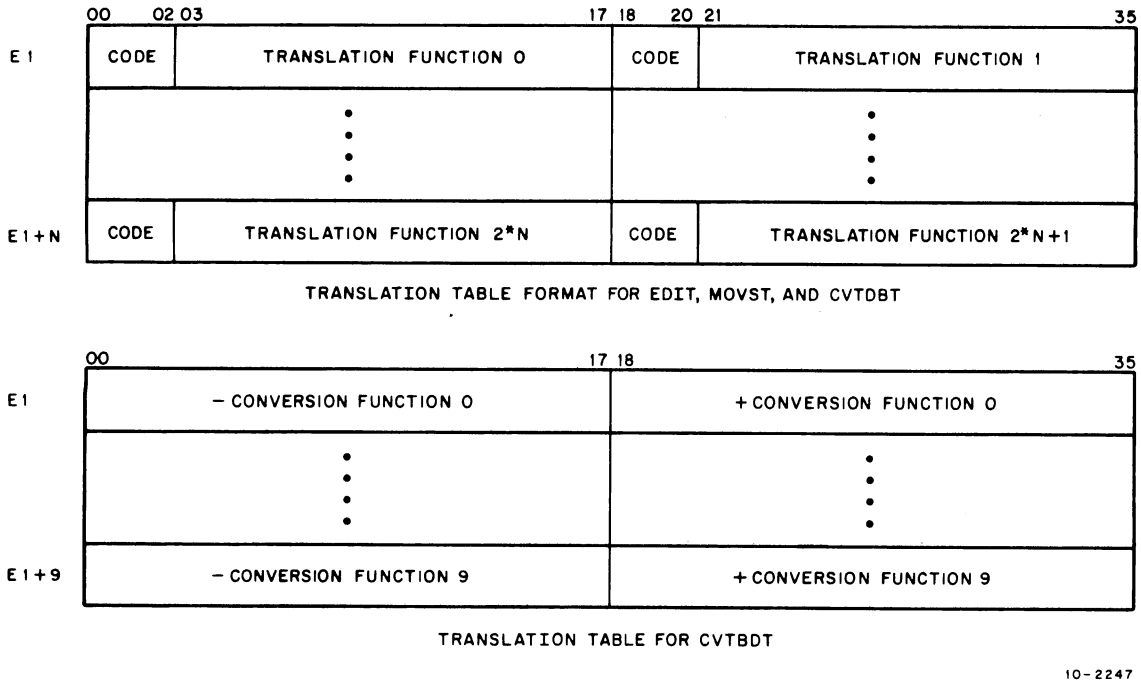


Figure 4 Translation Table Formats

MOVE STRING OPERATIONS

The move group provides the ability to move a byte string of specified length from an area of memory called the source, which is addressed by KL10 byte pointer to another area of memory called the destination, which is addressed by a second KL10 byte pointer.

Each of the four operators in the group provide a unique function that may be performed to the byte string as it is moved. The four operations together with the associated operators are illustrated in Table 2 as follows:

TABLE 2

MOVE STRING OPERATORS

OPERATOR	FUNCTION
MOVSLJ	LEFT JUSTIFICATION
MOVSRJ	RIGHT JUSTIFICATION
MOVSO	OFFSET
MOVST	TRANSLATION

LEFT JUSTIFICATION OVERVIEW

In MOVSLJ the byte string at the source is simply inserted in the destination. The three situations are illustrated in Figure 5.

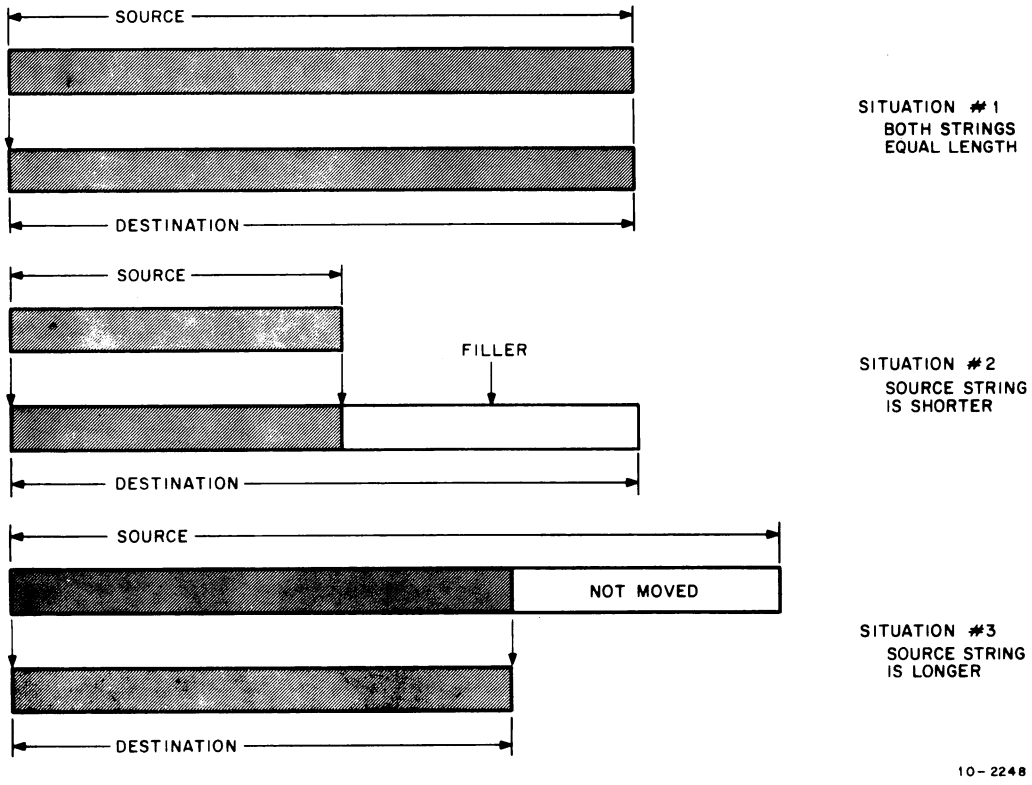


Figure 5 Move String with Left Justification (EXample)

In the first situation, the contents of the destination is simply replaced by the byte string from the source, and notice that left justification is inherent, since the byte pointer always points to the left most byte in both the source and destination.

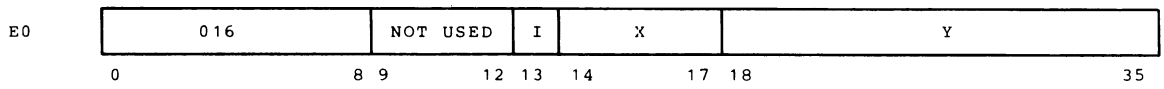
When the byte string at the source is shorter than the number of bytes required to fill the destination as in the case in situation #2, then after the source byte string is inserted in the destination

the remaining bytes of the destination receive the fill character obtained from location E0+1.

When the byte string of the source is longer than the number of bytes provided in the destination, which is the case in situation #3, then as much of the byte string at the source as will fit in the destination is moved and then the source is truncated, i.e., the remainder is not moved.

TECHNICAL SUMMARY (MOVSLJ)

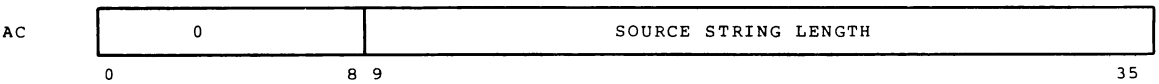
The operator MOVSLJ has the following format:



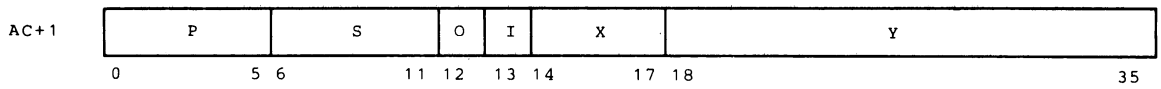
NOTE

Since the effective address is not used by MOVSLJ, the I and X fields should be made 0.

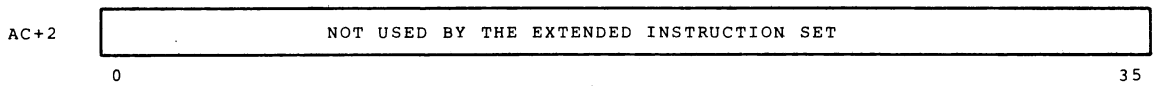
The following ACs are utilized by the MOVSLJ operator.



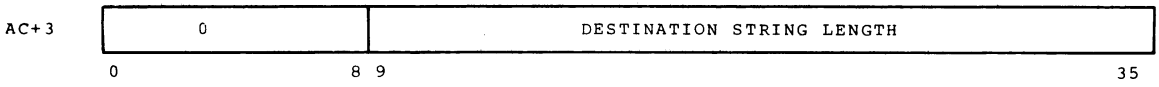
This represents the actual count of the number of bytes in the source byte string and is an unsigned number. If this number is 0, and the destination string length is non-zero, fill in the destination string with bytes taken from E0+1.



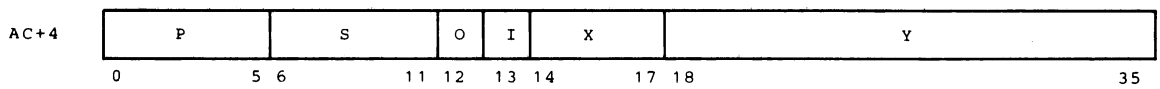
This is a standard KL10 byte pointer, and addresses the source byte string. See Section 2.3 for a description of byte manipulation.



This accumulator is not disturbed when the operator is performed.



This represents the actual count of the number of bytes in the destination string, and is an unsigned number. If this number is 0, no bytes are moved when the operator (MOVSLJ) is performed.



This is a standard KL10 byte pointer and addresses the destination byte string. See Section 2.3 for a description of byte manipulation.

LOGICAL FLOW (MOVSLJ)

If the destination length count contained in AC+3 is 0, go immediately to the next instruction without effecting the original contents of any ACs or memory operands in any way.

Otherwise, update the source and destination byte pointers contained in AC+1 and AC+4 and decrement the length counts contained in AC and AC+3. Now select a byte from the source byte string and place this byte in the destination string. Continue to move bytes in this fashion until either the source, destination, or both string lengths have been exhausted.

If the source length is the shorter, the remainder of the destination string is filled with bytes taken from E0+1.

If the source length is longer than the destination length, the destination string is truncated when the destination string length is exhausted, and the difference between the source and destination string lengths is stored in AC (NON-SKIP RETURN).

Upon completion of the instruction, the source byte pointer in AC+1 will be left addressing the last byte transferred from the source string. The destination byte pointer in AC+4 will be left addressing the last byte in the destination string into which a byte was moved. Now fetch the next instruction in the sequence and continue sequential operation from there (SKIP RETURN).

RIGHT JUSTIFICATION OVERVIEW

In the first situation illustrated on the next page, the contents of the destination is simply replaced by the byte string from the source.

In the second situation, the byte string at the source is shorter than the number of bytes required to fill the destination. In this case, the fill character, obtained from E0+1, is placed in leading bytes of the destination and the byte string from the source is then inserted to the right of the filler bytes. The number of filler characters used is equal to the difference between the number of bytes in the source and the number of bytes in the destination.

When the source string is longer than the number of bytes provided in the destination, which is the case in situation #3, leading bytes of the source are skipped over until the number of bytes remaining at the source is equal to the number of bytes in the destination.

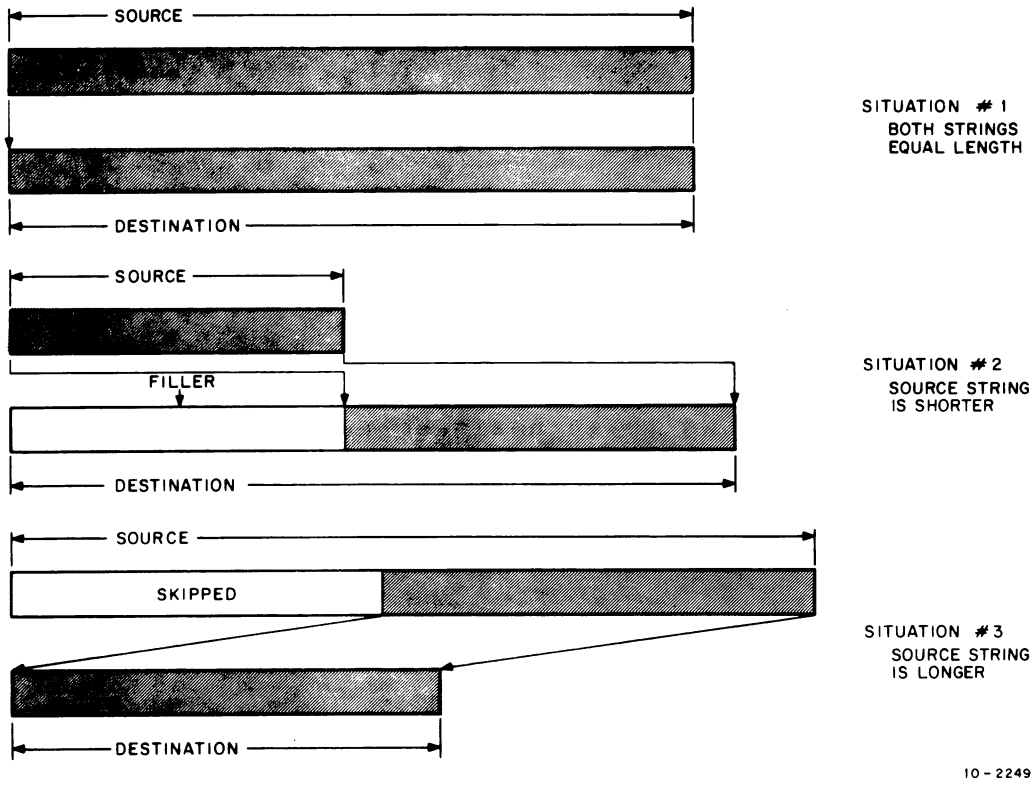
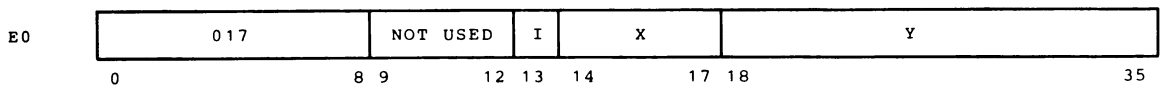


Figure 6 Move String with Right Justification (Example)

TECHNICAL SUMMARY (MOVSRJ)

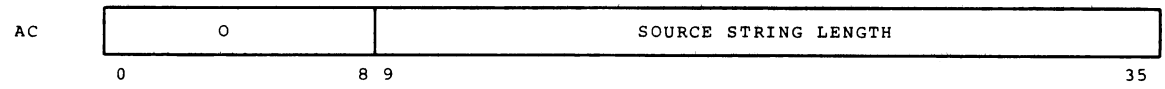
The operator MOVSRJ has the following format:



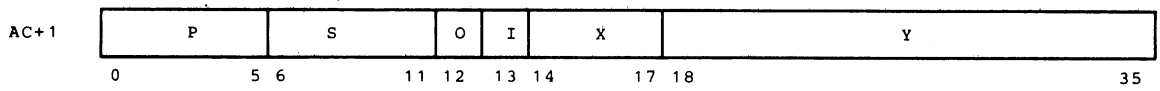
NOTE

Since the effective address is not used by MOVSRJ, the I and X fields should be made 0.

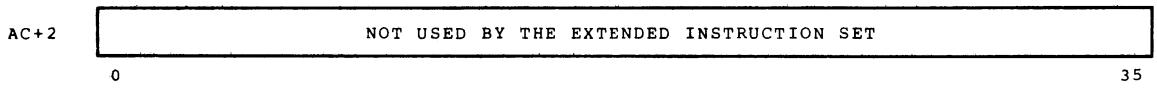
The following ACs are utilized by the MOVSRJ operator.



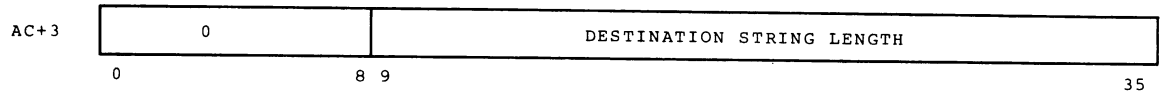
This represents the actual count of the number of bytes in the source byte string, and is an unsigned number. If this number is 0, and the destination string length is non-zero, fill in the destination string with bytes taken from E0+1.



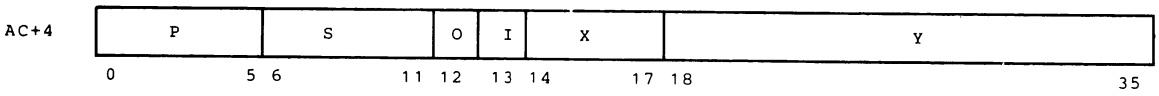
This is a standard KL10 byte pointer, and addresses the source byte string. See Section 2.3 for a description of byte manipulation.



This accumulator is not disturbed when the operator is performed.



This represents the actual count of the number of bytes in the destination string, and is an unsigned number. If this number is 0, no bytes are moved when the operator (MOVSRJ) is performed.



This is a standard KL10 byte pointer and addresses the destination byte string. See Section 2.3 for a description of byte manipulation.

LOGICAL FLOW (MOVSRJ)

If the destination length count contained in AC+3 is 0, go immediately to the next instruction without effecting the original contents of any ACs or memory operands in any way.

Otherwise, compute the difference between the initial source and initial destination length counts.

If the source byte string is the shorter, insert the fill character obtained from E0+1, in bytes of the destination string, updating the destination byte pointer contained in AC+4, and decrementing the destination length count contained in AC+3, until the number of fill characters inserted in the destination string is equal to the difference between the initial source and initial destination length counts.

But, if the source byte string is the longer, skip over bytes in the source string by updating the source byte pointer contained in AC+1, and decrementing the source length count contained in AC, until the number of bytes skipped over is equal to the difference between the initial source and initial destination length counts.

Now in any case, begin to move bytes from the source string utilizing both the source and destination byte pointers while decrementing both length counts. Continue moving bytes until the length counts are exhausted. Upon completion of the instruction, the source byte pointer in AC+1 will be left addressing the last byte transferred from the source string. The destination byte pointer in AC+4 will be left addressing the last byte in the destination string into which a byte was moved. Now skip the next instruction in the sequence and continue sequential operation from there.

OFFSET OVERVIEW

In MOVSO a byte string is moved from one area of memory to another. Each byte in the source string is OFFSET by adding the effective address E1, obtained from the operator word to it. For each byte the result is tested to see if the current size of the OFFSET byte is larger than that specified by the destination byte pointer.

If this is the case, moving the byte would effectively lose some information so the byte is not moved and the instruction is terminated. The source byte pointer, destination byte pointer, and length counts are stored in their respective ACs, and then the next instruction in the sequence is performed.

If each byte of the source with the offset added successfully passes the test, then upon completion of the instruction, the next instruction in the sequence is skipped. Figure 7 illustrates the three situations discussed above.

In the first situation, the source and destination string lengths are equal. Each byte of the source is offset, tested for proper fit in the destination string, and then placed in the destination string. Situation #2 is a case where the source string is shorter than the destination string. Bytes are obtained from the source, offset, and tested for proper fit in the destination string. When

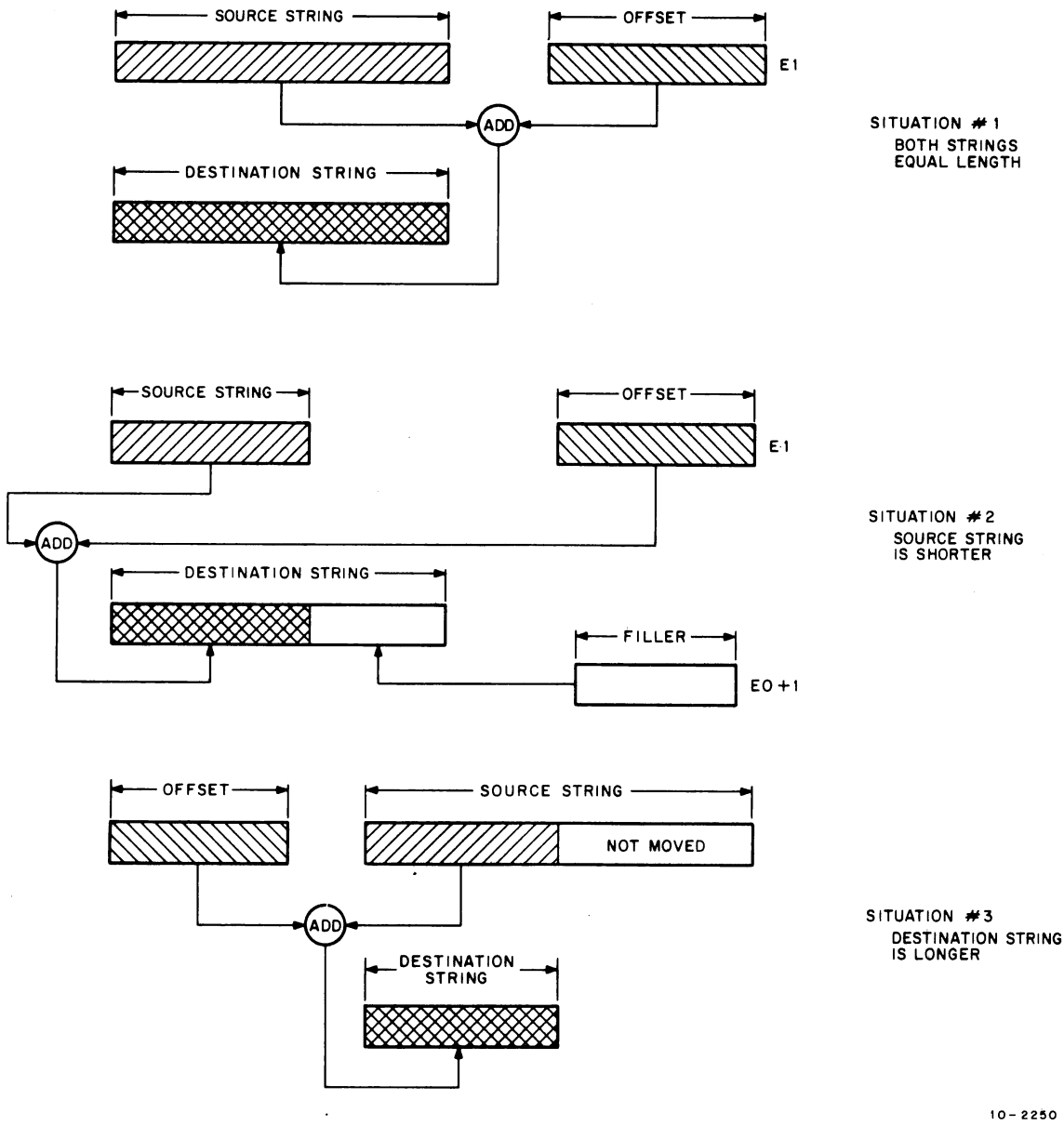
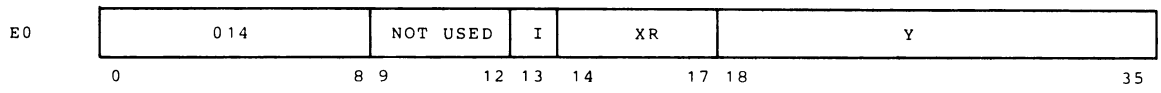


Figure 7 Offset Overview

the source length has been exhausted, the remainder of the destination string is filled with bytes taken from E0+1. If the source string is longer than the destination string, the destination string is truncated when the destination string length is exhausted, and the difference between the source and destination string length is stored in AC.

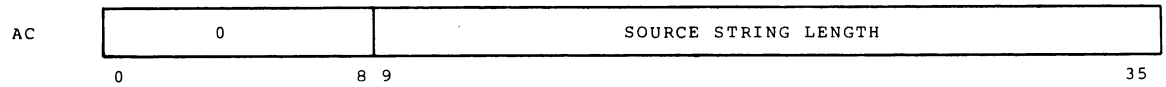
TECHNICAL SUMMARY (MOVSO)

The operator MOVSO has the following format:

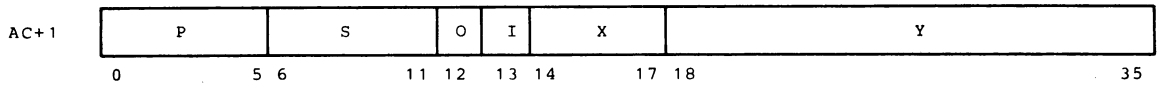


The effective address E1, which is calculated from I, X, and Y is the byte offset.

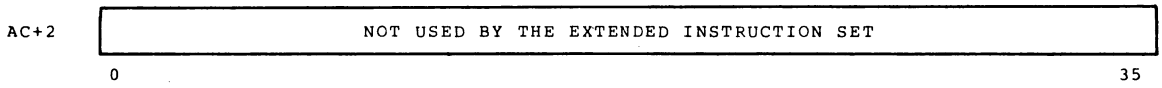
The following AC's are utilized by the MOVSO operator.



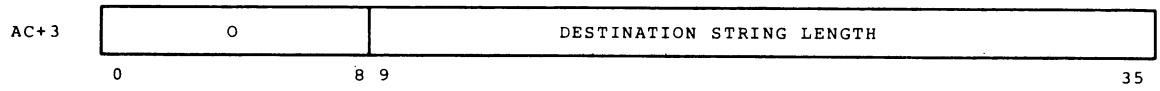
This represents the actual count of the number of bytes in the source string, and is an unsigned number. If this number is 0, and the destination string is non-zero, fill out the destination string with bytes taken from E0+1.



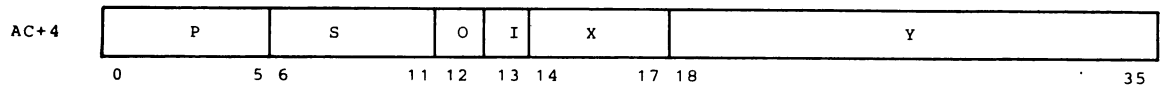
This is a standard KL10 byte pointer and addresses the source byte string. See Section 2.3 for a description of byte manipulations.



This accumulator is not disturbed when the operator is performed.



This represents the actual count of the number of bytes in the destination string, and is an unsigned number. If this number is 0, no bytes are moved when the operator (MOVSO) is performed.



This is a standard KL10 byte pointer and addresses the destination byte string. See Section 2.3 for a description of byte manipulation.

LOGICAL FLOW (MOVSO)

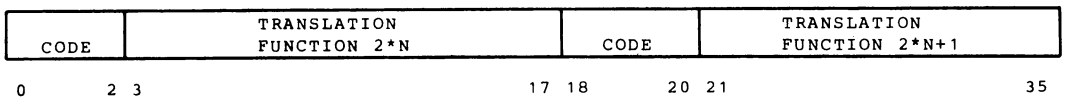
If the destination length count contained in AC+3 is 0, immediately skip the next instruction without effecting the original contents of any AC's or memory operands in any way. Otherwise, update the source byte pointer contained in AC+1, obtain a byte from the source string, and add the byte offset E1 to this byte. Unless all "1" bits of the sum are able to fit into the destination byte, the byte is not moved and the operation is terminated. The source byte pointer, destination byte pointer, and remaining length counts are stored in their respective AC's and then the next instruction in the sequence is performed. Otherwise, the destination byte pointer contained in AC+4 is updated, the byte is stored in the destination string, and, after each byte transfer, the source and destination length counts are decremented. When the source byte string is the shorter, the fill character is obtained from E0+1 and placed in the remaining bytes of the destination string; updating the destination byte pointer contained in AC+4, and decrementing the destination length count countained in AC+3 until the destination length count is exhausted. If the source byte string is the longer, the destination string is truncated when the

destination length count is exhausted, and the difference between the source and destination string lengths is stored in AC. Upon completion of the instruction, the source byte pointer in AC+1 will be left addressing the last byte transferred from the source string. The destination byte pointer in AC+4 will be left addressing the last byte in the destination string into which a byte was moved. Now skip the next instruction in the sequence and continue sequential operation from there.

TRANSLATION OVERVIEW

In MOVST a byte string is moved from one area of memory to another. Each byte in the source string is TRANSLATED through a table whose base address is E1. The source byte is an Index which is added to the base E1, and the result is used to address a halfword entry in the table. The CODE in the high order three bits indicates the disposition of the byte. The code allows for aborting the operation with or without setting flags in AC or starting significance while setting flags in AC.

The translation table format is as follows:



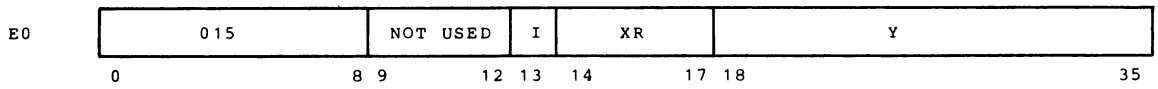
CODE	FUNCTION
0	NOP
1	Abort
2	Clear the M FLAG in AC.
3	Set the M FLAG in AC.
4	Set both the S and N Flag.
5	Set the N FLAG in AC and then Abort.
6	Clear the M FLAG in AC and then set both the S and N flags in AC.
7	Set the M FLAG in AC and then set both the S and N flags in AC.

Bit 0 of AC is called the S or significance flag. As the source string is scanned from left to right, each byte is translated. As long as the S Flag is 0, no reference is made to the destination string. The S Flag may be set by the programmer when the instruction is started, or may be set by the instruction upon detection of

the significance starter bit (bit 0 or 18 respectively) in the translation table. Bit 1 of AC is called the N or non-zero flag. It may be set by the programmer when the instruction is started, or may be set by the instruction upon detection of the significance starter bit (bit 0 or 18 respectively) in the translation table. Bit 2 of AC is called the M or minus flag. It may be set by the programmer and may be set, cleared, or left alone by the sign control bits (CODE) in the translation table.

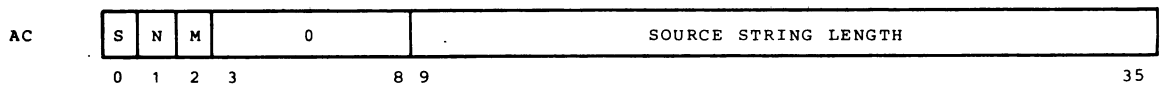
TECHNICAL SUMMARY (MOVST)

The operator MOVST has the following format:



The effective address E1, which is calculated from I, X, and Y, is the base address of the translation table.

The following AC's are utilized by the MOVST operator.

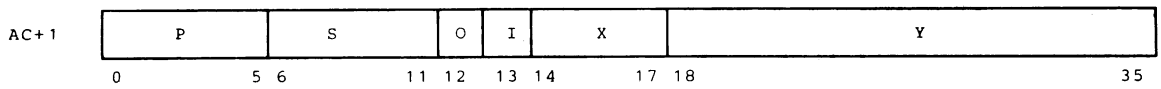


Bits 0-2 of AC constitute the translation flags. The meaning of each bit is as follows:

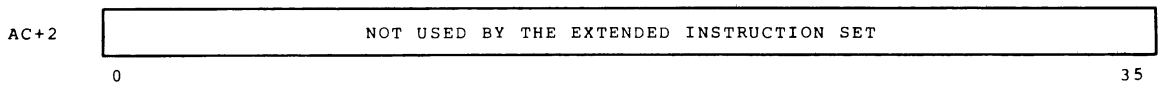
S FLAG - SIGNIFICANCE FLAG. This flag is checked by the instruction for each byte being translated. If the S Flag in AC is equal to a 1, the byte obtained from the translation table is placed in the destination string, except in the case where the translation table entry contained the ABORT CODE, in which case the destination string is not referenced and the operation is terminated. If the S Flag in AC is equal to 0, the byte being translated is not significant, so the destination string is not referenced (this assumes that the S bit in the translation table code, bit 0 is equal to 0). If the S bit in the translation table code is equal to a 1 for the byte being translated, the instruction sets both the S Flag in AC and the N Flag in AC.

N FLAG - NON-ZERO FLAG. This flag has no direct effect on the instruction. The N Flag is set by the instruction any time the significance starter bit is on in the translation table (Codes 4, 6, and 7) and also by a code of 5 which sets N and then aborts but does not set S.

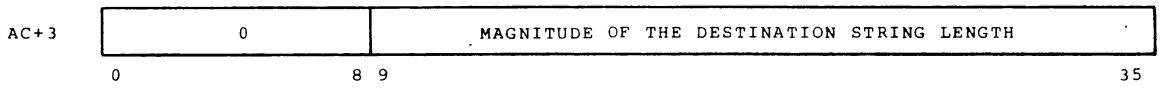
M FLAG - MINUS FLAG. This flag may be set or cleared by the programmer and set, cleared, or left alone by the sign control bits (Codes 2, 3, 6, and 7) in the translate table.



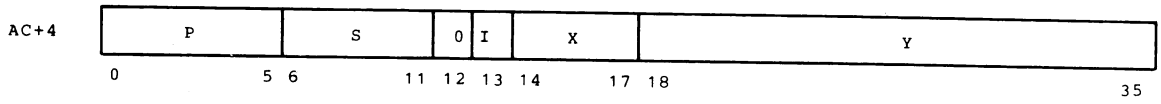
This is a standard KL10 byte pointer, and addresses the source byte string. See Section 2.3 for a description of byte manipulation.



This accumulator is not disturbed when the operator is performed.



This represents the actual count of the number of bytes in the destination string and is an unsigned number. If this number is 0, no bytes are moved when the operator (MOVST) is performed.



This is a standard KL10 byte pointer and addresses the destination byte string. See Section 2.3 for a description of byte manipulation.

LOGICAL FLOW (MOVST)

If the destination length count contained in AC+3 is 0, immediately skip the next instruction without effecting the original contents of any AC's or memory operands in any way. If the source length count contained in AC is 0, and the destination length count contained in AC+3 is non-zero, place the contents of E0+1 in all remaining bytes of the destination string; updating the destination byte pointer contained in AC+4, and decrementing the length count contained in AC+3 until the destination length count is exhausted. Otherwise, update the source byte pointer contained in AC+1, obtain a byte from the source string, and using this byte as an index, obtain a halfword from the translation table; selecting the left halfword if the source byte is even and the right halfword if the source byte is odd. Now evaluate the three high order bits of this entry together with the most significant bit of AC (S Flag), as described above. Continue to translate bytes in this fashion until either the source, destination, or both length counts are exhausted,

or an abort code is detected in the halfword. If an abort code is detected, the operation is terminated and the next instruction in the sequence is fetched. In this situation the last source byte referenced was translated, but no reference was made to the destination string for this byte. If the source byte string is the longer, the destination string is truncated when the destination length count is exhausted, and the difference between the source and destination lengths is stored in AC. Upon completion of the instruction, the source byte pointer in AC+1 will be left addressing the last byte transferred from the source string. The destination byte pointer in AC+4 will be left addressing the last byte in the destination string into which a byte was moved. Now skip the next instruction in the sequence and continue sequential operation from there.

COMPARE STRING OPERATIONS

The compare group provides the ability to compare a string of specified length from an area of memory called the source, which is addressed by a KL10 byte pointer with another string also of specified length from an area of memory called the destination, which is addressed by a second KL10 byte pointer. A skip or non-skip feature is provided by a set of six possible compare operators. The six operators, together with their associated functions, are illustrated in Table 3 as follows:

TABLE 3

COMPARE STRING OPERATORS

OPERATOR	OPCODE	FUNCTION
CMPSL	001	Compare the source string with the destination string, and skip the next instruction the sequence if the source string is less than the destination string.
CMPSE	002	Compare the source string with the destination string, and skip the next instruction in the sequence if the magnitudes of both strings are equal.
CMPSLE	003	Compare the source string with the destination string, and skip the next instruction in the sequence if the magnitude of the source string is less than or equal to the magnitude of the destination string.
CMPSGE	005	Compare the source string with the destination string, and skip the next instruction in the sequence if the magnitude of the source string is greater than or equal to the magnitude of the destination string.

TABLE 3 (Cont.)

COMPARE STRING OPERATORS

OPERATOR	OPCODE	
CMPSN	006	Compare the source string with the destination string, and skip the next instruction in the sequence if the strings are unequal in magnitude.
CMPSG	007	Compare the source string with the destination string, and skip the next instruction in the sequence if the magnitude of the source is greater than the magnitude of the destination string.

COMPARE OVERVIEW

In CMPSXX bytes are alwayw compared as unsigned numbers.

The relationship between the two strings being compared is determined by the relationship of the first pair of differing bytes encountered. This is to say that if the first n bytes of the source string are equal in magnitude to the first n bytes of the destination string, but the n+1th byte of the source string is greater than the n+1th byte of the destination string then by definition,

the source string is greater. If one of the strings is shorter than the other, a filler is used for subsequent comparisons, when the shorter string runs out. When the source string is the shorter the contents of E0+1 is used as the filler and when the destination string is the shorter the contents of E0+2 is used as the filler. The three situations are illustrated below.

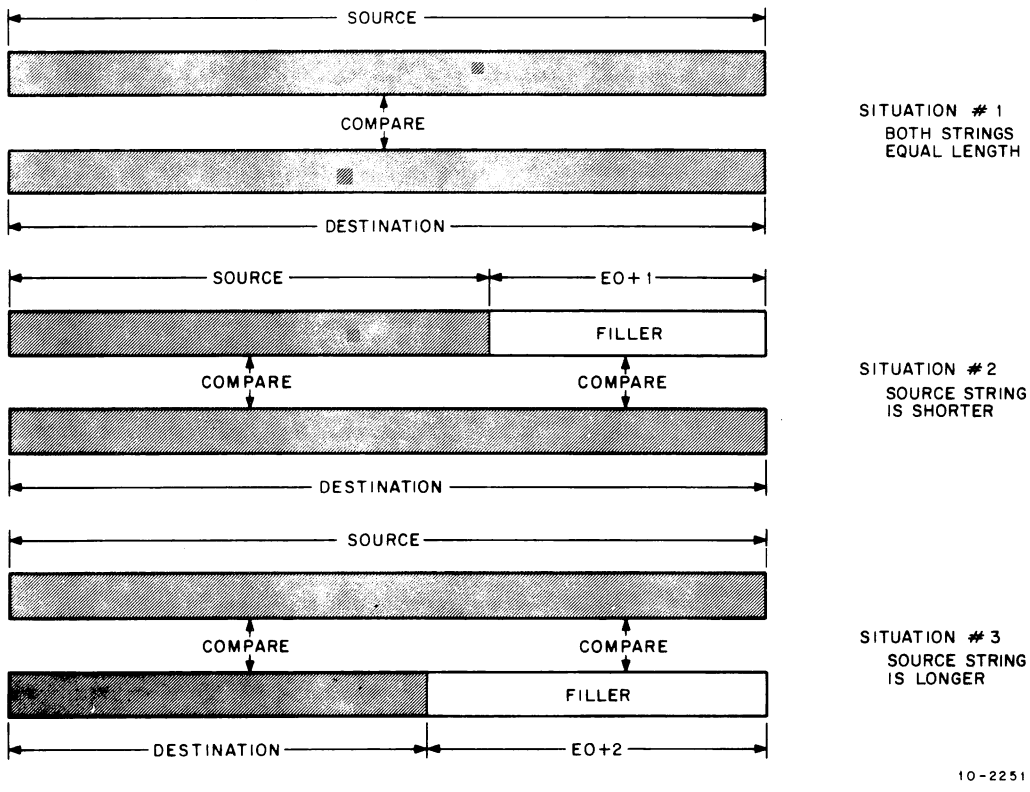
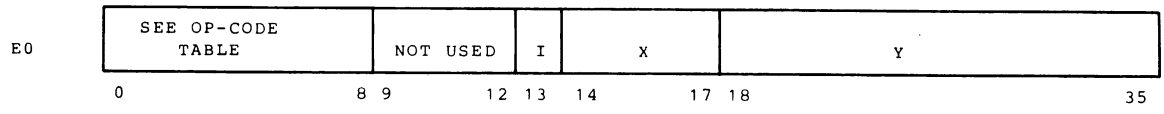


Figure 8 Compare Strings (Example)

In Situation #1, the source and destination strings are of equal length. Bytes taken from the two strings must be compared until an inequality is found, or the length counts are exhausted. If the byte string at the source is shorter, as in Situation #2, bytes are compared from both strings until either an inequality is found or the source length count is exhausted. If the source length is exhausted first, then for the remainder of the comparison, the contents of E0+1 is compared with the remaining bytes in the destination string. Similarly, if the byte string at the source is longer, as in Situation #3, bytes are compared from both strings until either an inequality is found or the destination length count is exhausted. If the destination length is exhausted first, then for the remainder of the comparison, the contents of E0+2 is compared with the remaining bytes in the source string.

TECHNICAL SUMMARY (CMPSXX)

The set of 6 compare operators have the following format:



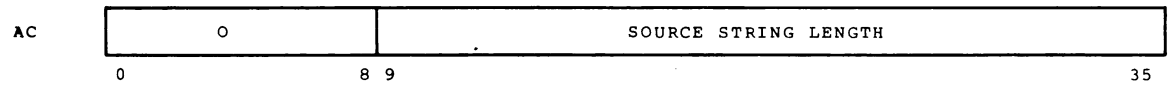
OP-CODE TABLE

MNEMONIC	OP-CODE
CMPSL	001
CMPSE	002
CMPSLG	003
CMPSGE	005
CMPSN	006
CMPSG	007

NOTE

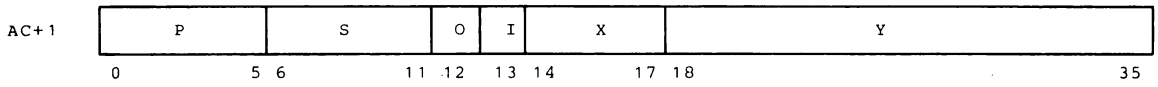
Since the effective address is not used by CMPXX, the I and X fields should be made 0.

The following AC's are utilized by the CMPSXX operators.

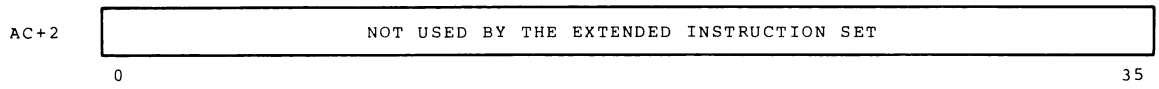


This represents the actual count of the number of bytes in the source byte string, and is an unsigned number. If this number is 0, and the destination string length is non-zero, the contents of

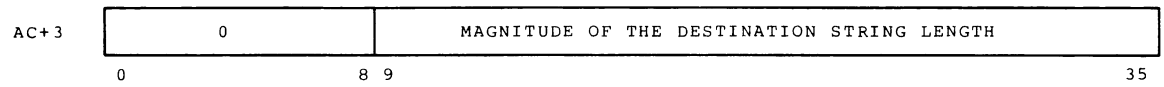
E0+1 is compared with successive bytes taken from the destination string.



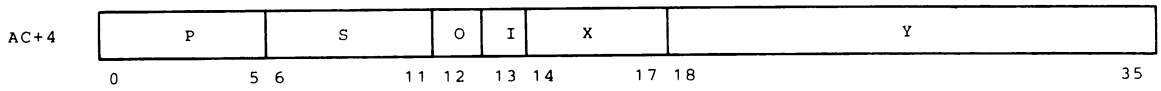
This is a standard KL10 byte pointer, and addresses the source byte string. See Section 2.3 for a description of byte manipulation.



This accumulator is not disturbed when the operator is performed.



This represents the actual count of the number of bytes in the destination string, and is an unsigned number. If this number is 0, and the source string length is non-zero, the contents of E0+2 is compared with successive bytes taken from the source string.



This is a standard KL10 byte pointer and addresses the destination string. See Section 2.3 for a description of byte manipulation.

DECIMAL AND BINARY OPERATIONS

The convert group provides the ability to convert binary integers represented in KL10 double precision format to a string of decimal digits specified by a standard KL10 byte pointer and of a prescribed length. In addition, a string of decimal digits specified by a standard KL10 byte pointer and of a prescribed length may be converted to a double word binary integer in KL10 double precision format. The group consists of four operators illustrated below in Table 4.

TABLE 4

DECIMAL AND BINARY OPERATORS

OPERATOR	OP CODE	FUNCTION
CVTDBO	010	OFFSET the byte string at the source converting resulting decimal digits to binary.
CVTDBT	011	TRANSLATE the byte string at the source converting resulting decimal digits to binary.

TABLE 4 (Cont.)

DECIMAL AND BINARY OPERATORS

OPERATOR	OP CODE	FUNCTION
CVTBDO	012	Convert the double word binary at the source to a decimal digit string offsetting converted digits.
CVTBDT	013	Convert the double word binary at the source to a decimal digit string translating converted digits.

DECIMAL TO BINARY CONVERSION OVERVIEW

In both CVTDBO and CVTDBT, a source string, which is addressed by a standard KL10 byte pointer, is processed as a string of decimal digits. Before beginning the conversion, the S flag in AC is tested to determine whether to initialize the double word result registers AC+3, AC+4. These registers are initially cleared only if the S flag in AC is zero. This makes it possible for the programmer to continue a conversion which was stopped by detection of a non-digit. For CVTDBO, when conversion begins, the S flag in AC is set equal to 1, indicating a conversion has been started. There are essentially two steps involved in processing bytes with either CVTDBO or CVTDBT.

OFFSET

The first step in CVTDBO is to add the OFFSET E1 to the source byte and then check the result to determine if it is a decimal digit between 0 and 9. When the addition of the OFFSET results in a non-digit the operation is immediately terminated. In this situation, the destination is not referenced and the non-skip return is made, i.e., the next instruction in the program is performed.

TRANSLATED

The first step in CVTDBO is to obtain the translation function from the table. To do this, the source byte is used as an index into the translation table whose base is E1. The format for entries is the same as that covered under MOVST.

Next, the code in the high-order three bits of the translation function is evaluated. Codes of 4, 6, or 7 have the significance starter bit turned on, as a result, the S flag in AC is set and AC+3 and AC+4 is not cleared. In addition, these three codes manipulate other flags in AC. Codes of 1 or 5 abort the operation but also do not clear AC+3 and AC+4. A code of 3 sets the M flag only and a code of 0 sets nothing. Now if the S flag in AC is set or becomes set, the low order four bits of the translation function are evaluated to determine whether they constitute a digit between 0 and 9. If they do not, the operation is terminated (this

is also the case for the abort codes 1 or 5), the destination is not referenced and the non-skip return is made.

COMMON OPERATIONS

For either CVTDBO or CVTDBT, the second step is the same, here the instruction converts the decimal digit to binary form, multiplies the entire double word register AC+3 and AC+4 by 10 decimal and then adds in the binary. Consider the 3 digit example illustrated in Figure 9.

CONVERT 139_{10} TO BINARY

AC+3	AC+4	CONVERSION STEP
000000,000000	000000,000000	S FLAG = 0 INITIALLY CLEAR AC+3 AND AC+4
000000,000000	000000,000000	MULTIPLY BY 10_{10}
000000,000000	000000,000001	ADD IN DECIMAL DIGIT (1)
000000,000000	000000,000012	MULTIPLY BY 10_{10}
000000,000000	000000,000015	ADD IN DECIMAL DIGIT (3)
000000,000000	000000,000202	MULTIPLY BY 10_{10}
000000,000000	000000,000213	ADD IN DECIMAL DIGIT (9) OBTAIN RESULT

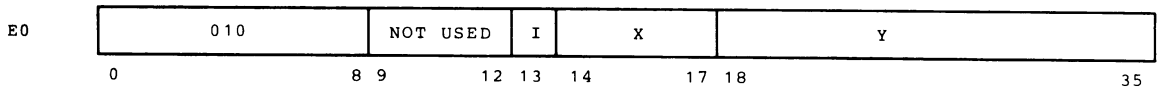
10-2252

Figure 9 Decimal to Binary Conversion (Example)

Thus in the example, the result is 213_8. Note that if the S flag had been set initially, the contents of AC+3 and AC+4 would have been multiplied by 10 decimal prior to adding in the first digit. If the entire string is converted successfully and the M flag is set in AC, the instruction negates the doubleword result.

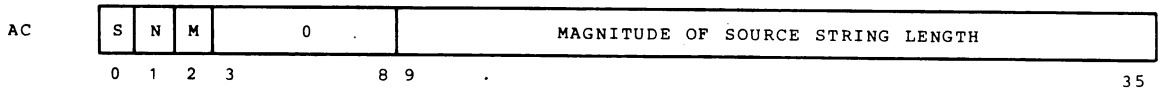
TECHNICAL SUMMARY (CVTDBO)

The operator CVTDBO has the following format:



The effective address E1, which is calculated from I, X, and Y is the byte OFFSET.

The following AC's are utilized by the CVTDBO operator.



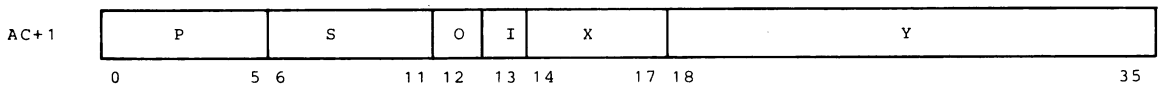
S FLAG - Significance. This flag is checked by the instruction for each byte being converted. If the S flag in AC is equal to a 1, the double length number con-

tained in AC+3 and AC+4 is multiplied by 10 decimal prior to adding in the converted bits of the binary integer formed by adding the offset to the source byte.

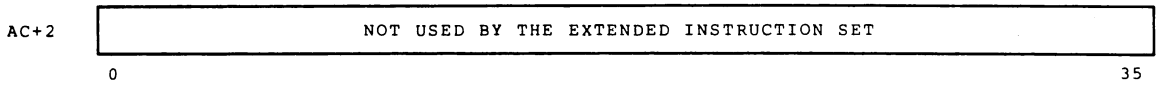
N FLAG - Non-Zero. This flag is not used during CVTDBO.

M FLAG - Minus Flag. This flag may be set by the programmer. If it is equal to a 1, before the conversion begins, the resulting binary integer result in AC+3 and AC+4 is negated upon successful termination of the instruction.

Bits 9-35 represent the actual count of the number of bytes in the source string, and is an unsigned number. If this number is 0, no bytes are converted when the operator (CVTDBO) is performed.

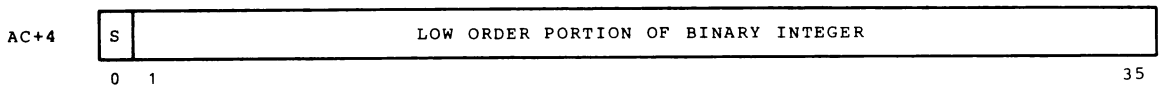


This is a standard KL10 byte pointer and addresses the source byte string. Seee Section 2.3 for a description of byte manipulations.



This accumulator is not disturbed when the operator is performed.

Bit 0 of AC+3 is the sign of the two part binary integer contained in AC+3 and AC+4. If it is 1, the integer is a two's complement negative number. If it is 0, the integer is positive.



Bit 0 of AC+4 is a copy of the sign bit (bit 0 of AC+3).

LOGICAL FLOW (CVTDBO)

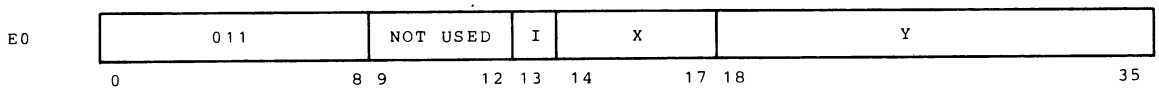
Decrement the source length count contained in AC and update the source byte pointer contained in AC+1. If the S Flag (bit 0) in AC is equal to zero, clear AC+3 and AC+4; then set the S Flag in AC, otherwise do not clear AC+3 and AC+4. Now obtain a byte from the specified source string adding the OFFSET E1 and test the low order four bits of the result. If these low order four bits are

not strictly between 0 and 9, no further bytes are converted, the remaining length count with flags (S, N, M) are stored in AC, and the next instruction in the sequence is fetched. Otherwise, multiply the double length number contained in AC+3 and AC+4 by 10 decimal and add in the converted binary bits of the source byte.

Continue conversion until either the source length count is exhausted, i.e., becomes 0, or a non-digit is detected. In the case where the length count is exhausted, test the M Flag (bit 2) of AC and if equal to 1, negate the double length result contained in AC+3 and AC+4. Upon completion of the instruction, the source byte pointer in AC+1 will be left addressing the last byte converted. Now skip the next instruction in the sequence and continue sequential operation from there.

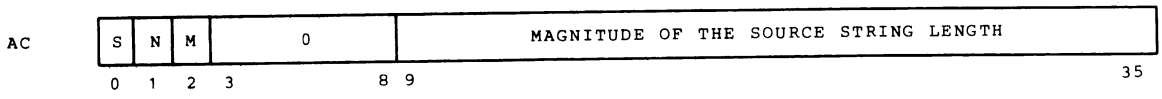
TECHNICAL SUMMARY (CVTDBT)

The operator CVTDBT has the following format:



The effective address E1, which is calculated from I, X, and Y is the base address of the translation table.

The following AC's are utilized by the CVTDBT operator.



Bits 0-2 of AC constitute the translation flags. The meaning of each bit is as follows:

S FLAG - Significance Flag. This flag is checked by the instruction for each byte being converted. If the S flag in AC is equal to 1, the double length number contained in AC+3 and AC+4 is multiplied by 10 decimal prior to adding in the bits of the binary integer formed by converting the low order four bits of the halfword obtained from the translation table.

If the S flag in AC is 0, both AC+3 and AC+4 are cleared before beginning the conversion.

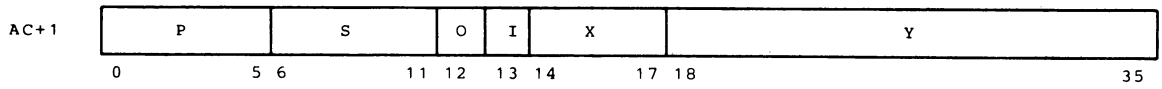
N FLAG - Non-Zero. This flag is set by the instruction if the significance starter bit is on in a table halfword (CODE 4, 6, or 7).

M FLAG - Minus Flag. This flag may be set by the programmer or during translation. If upon successful termina-

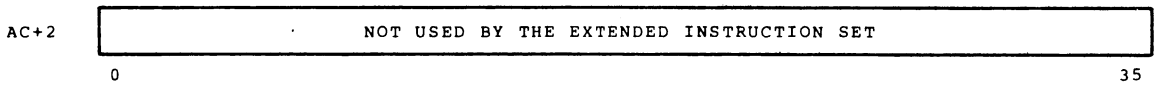
tion of the instruction the M flag is a 1, the resulting binary integer in AC+3 and AC+4 is negated. negated.

Bits 9-35 represent the actual count of the number of bytes in the source string and is an unsigned number.

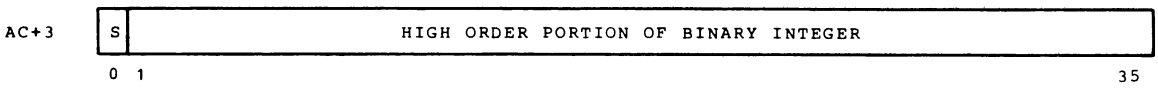
If this number is 0, no bytes are converted when the operator (CVTDBT) is performed.



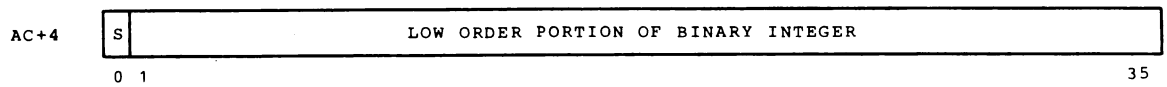
This is a standard KL20 byte pointer and addressed the source byte string. See Section 2.3 for a description of byte manipulation.



This accumulator is not disturbed when the operator is performed.



Bit 0 of AC+3 is the sign of the two-part binary integer contained in AC+3 and AC+4. If it is 1, the integer is a two's complement negative number. If it is 0, the integer is positive.



Bit 0 of AC+4 is a copy of the sign bit (bit 0 of AC+3).

LOGICAL FLOW (CVTDBT)

Decrement the source length count contained in AC, and update the source byte pointer contained in AC+1; then obtain a byte from the source string and using this byte as an index obtain a halfword translation function from the table, selecting the left halfword entry if the source byte is even, or the right halfword entry if the source byte is odd. Now evaluate the three high order bits of the selected entry, as described above. If the significance starter bit in the halfword is on, set the S and N flag S in AC begin conversion by testing low order four bits of the translation function. If these bits are not strictly between 0 and 9, no further bytes are converted, the remaining length count with flags (S, N, M) are stored in AC, and the next instruction in the sequence is fetched. Otherwise, multiply the double length number contained in AC+3 and

AC+4 by 10 decimal and add the low order four bits of the translation function to the result. Continue the conversion until the length count is exhausted, i.e., becomes 0 or an abort code is detected in the table. The action taken when an abort code is detected is the same as that for a non-digit. When the length count has been exhausted, test the M flag (bit 2) of AC and if equal to 1, negate the double length result contained in AC+3 and AC+4. Upon completion of the instruction, the source byte pointer in AC+1 will be left addressing the last byte converted. Now skip the next instruction in the sequence and continue sequential operation from there.

BINARY TO DECIMAL CONVERSION OVERVIEW

In both CVTDBO and CVTBDT, the magnitude of a 70 bit double word binary integer contained in AC and AC+1 is processed to produce a decimal digit string which is stored as specified by a standard KL10 byte pointer.

CONVERSION PROCESS

The Conversion Process produces a decimal integer result, most significant digit first. As many as 22 digits may be generated from the 70 bit magnitude portion of the binary integer. The instruction begins the conversion by finding the first power of 10, which is greater than the binary integer up to a value of 10^{21}. When the binary integer N has the range $2^{70} \leq n < 2^{71}$, the most significant digit of the result is guaranteed to be a 1. In this situa-

tion, the digit 1 is generated as the most significant digit of the result, and the binary integer is compensated for the effective extraction of the MSD by subtraction of 10^{21} from it. For binary integer values less than 2^{70}, the power of 10 formed is equivalent to the number of decimal digits required to represent the binary integer. For integer values equal to, or greater than, 2^{70}, the power of 10 has a value of 21 and the 22nd digit (MSD) is known to be a 1. Next, the binary integer is divided by the power of 10 producing a fractional result. This is always the case since the divisior is always larger. The resulting binary fraction is approximate in many cases (truncated) and the instruction includes a correction step to handle this situation. The error in the faction is never greater than 2^{-71} and the quotient is never larger than the exact binary representation desired. The instruction therefore adds 2^{-71} to the quotient which guarantees that the result is now larger than the desired fraction but with an error not greater than 2^{-71}. The instruction now forms significant decimal digits by multiplying the fraction by 10 and extracting the high order 4 bits then shifting the faction and multiplying once again. This process is continued until the required number of decimal digits have been formed.

PRELIMINARY

If the binary integer in AC and AC+1 is negative, it is made positive and the M flag is set before beginning the conversion. The

instruction then calculates the number of decimal digits required to represent the binary integer and compares this with the destination length count contained in AC+3. If the length count is less than that calculated, the instruction is immediately terminated and the non-skip return is made. When the length count is greater than the number of digits calculated for the result, the S flag in AC+3 is tested. If the S flag is equal to 1, (E0+1) is stored in successive bytes of the destination string to right-align the decimal integer result. If the S flag is equal to 0, or the length count and calculated number of digits are equal. The conversion begins with the first significant digit of the binary integer.

OFFSET

In CVTBDO the OFFSET E1 is added to each digit formed and the result is stored in the destination string. Upon completion, the instruction sets the S flag in AC+3 and clears AC and AC+1. If the binary integer was non-zero, the N flag will be set; if it was negative, the M flag will be set.

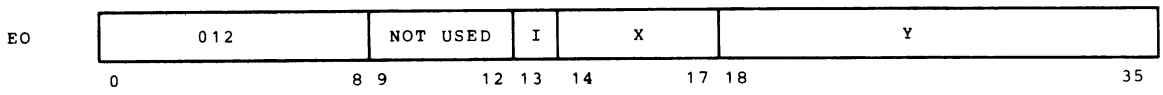
TRANSLATED

In CVTBDT, the conversion function is obtained from the translation table. The table is different than that for MOVST. The byte representations of each digit are in a ten-word table whose base address is E1. The right half of each table entry is a "Positive

Digit", and the left half is the corresponding "Negative Digit". The Positive digits are used for all but the low order digit of the result, and the low order digit is taken from the left half entry if the M flag is equal to 1 and from the right half entry if the M flag is equal to 0. Upon completion, the instruction sets the S flag in AC+3 and clears AC and AC+1. Same as above.

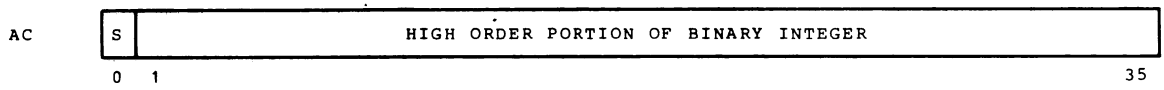
TECHNICAL SUMMARY (CVTBDO)

The operator CVTBDO has the following format:



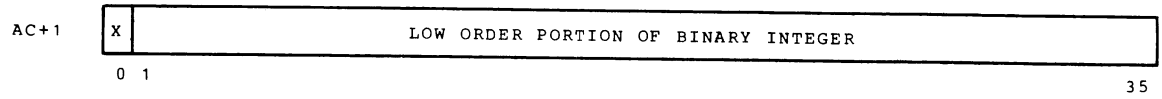
The effective address E1, which is calculated from I, X, and Y is the byte OFFSET.

The following AC's are utilized by the CVTBDO operator.

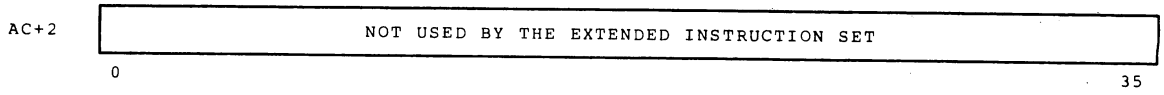


Bit 0 of AC is the sign of the two-part binary integer contained in AC and AC+1. If it is 1, the integer is a two's complement

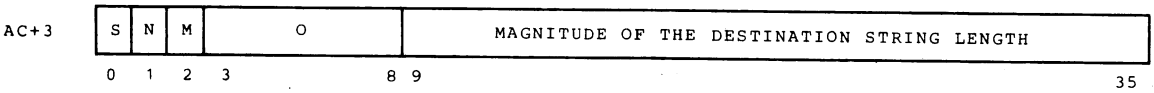
negative number. If it is 0, the integer is positive.



Bit 0 of AC+1 is ignored by the CVTDBO instruction.



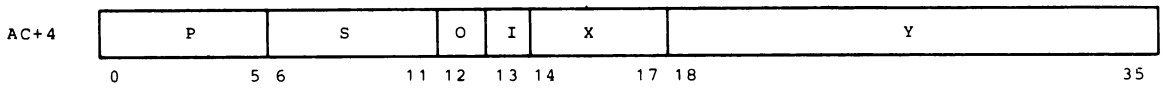
This accumulator is not disturbed when the operator is performed.



S FLAG - Significance. This flag is checked by the instruction prior to beginning the conversion. If the S Flag in AC+3 is equal to 1 and the length count contained in AC+3 is greater than the number of digits required to represent the binary integer contained in AC and AC+1, then the contents of location E)+1 is placed in successive bytes of the destination string, right-aligning the decimal digits of the result.

N FLAG - Non-zero Flag. If significance is detected in the binary source, the N flag in AC+3 is set by the instruction.

M FLAG - Minus Flag. If the sign of the high order portion of the binary integer contained in AC is equal to 1, the M flag is set in AC+3 by the instruction.



This is a standard KL10 byte pointer and addresses the destination string. See Section 2.3 for a description of byte manipulation.

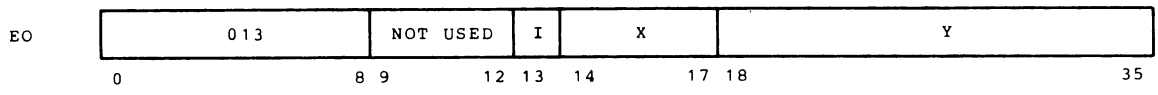
LOGICAL FLOW (CVTBDO)

Evaluate the binary integer contained in AC and AC+1, determining the number of deciaml digits required to represent the magnitude of the binary integer, and compare this with the length count contained in AC+3. If the length count is less than the number of decimal digits required, go immediately to the next instruction without effecting the original contents of any AC's or memory operands in any way. If the length count is greater than the number of decimal digits required and the S flag in AC+3 (Bit 0) is equal to 1, place the contents of E0+1 in successive bytes of

the destination string specified by the byte pointer contained in AC+4, to right-align the decimal digits of the result. Otherwise, if the S flag is equal to 0 or the length count is equivalent to the number of decimal digits required, begin conversion by computing the first significant digit in the binary integer, adding the byte offset E1 to the decimal digit obtained. Continue conversion by updating both the length count contained in AC+3 and the byte pointer contained in AC+4 until the length count is exhausted. The byte pointer contained in AC+4 is left addressing the last digit stored in the destination string (low order digit), the length count is zero, AC and AC+1 contain zero. Now skip the next instruction in the sequence and continue sequential operation from there.

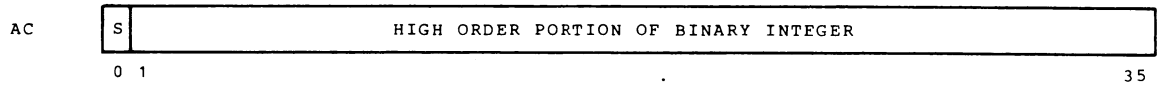
TECHNICAL SUMMARY (CVTBDT)

The operator CVTBDT has the following format:

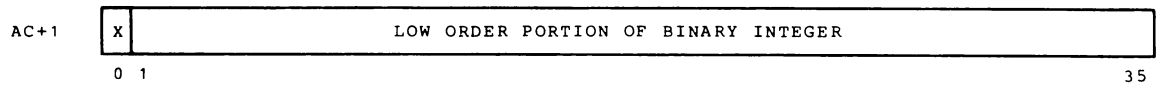


The effective address E1, which is calculated from I, X, and Y is the base address of the translation table.

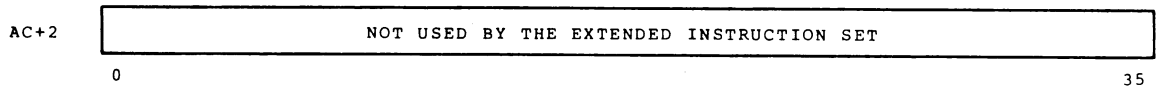
The following AC's are utilized by the CVTBDT operator.



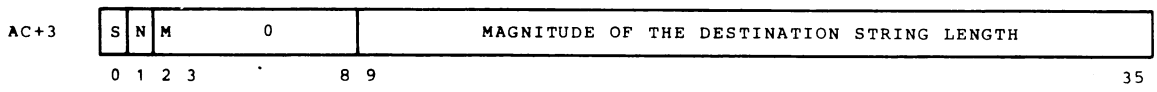
Bit 0 of AC is the sign of the two-part binary integer contained in AC and AC+1. If it is 1, the integer is a two's complement negative number. If it is 0, the integer is positive.



Bit 0 of AC+1 is ignored by the CVTBDT instruction.



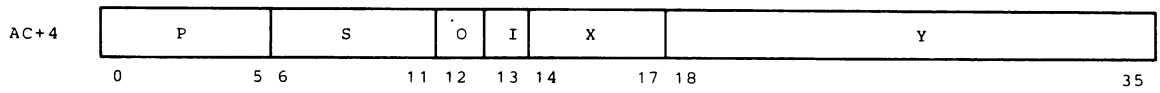
This accumulator is not disturbed when the operator is performed.



S FLAG - Significance. This Flag is checked by the instruction prior to beginning the conversion. If the S Flag in AC+3 is equal to 1 and the length count contained in AC+3 is greater than the number of digits required to represent the binary integer contained in AC and AC+1, then the contents of location E0+1 is placed in successive bytes of the destination sting, right-aligning the decimal digits of the result.

N FLAG - Non-Zero Flag. If AC and AC+1 are non-zero, the N Flag in AC+3 is set by the instruction.

M FLAG - Minus Flag. If the sign of the integer contained in AC is equal to 1, the M Flag is set in AC+3 by the instruction.



This is a standard KL10 byte pointer and addresses the destination string. See Section 2.3 for a description of byte manipulation.

LOGICAL FLOW (CVTBDT)

Evaluate the binary integer contained in AC and AC+1 determining the number of decimal digits required to represent the magnitude of the binary inter, and compare this with the length count contained in AC+3. If the length count is less than the number of decimal digits required, go immediately to the next instruction without effecting the original contents of any AC's or memory operands in any way. If the length count is greater than the number of decimal digits required and the S flag in AC+3 (Bit 0) is equal to 1, then place the contents of E0+1 in successive bytes of the destination string specified by the byte pointer contained in AC+4 to right-align the decimal digits of the result. Otherwise, if the S flag is equal to 0 or the length count is equivalent to the number of decimal digits required, begin conversion by computing the first significant digit in the binary integer; and using this digit as an index, obtain the half-word byte representation of the digit from the translation table, selecting the right half-word (positive digit) from the table for all but the low order digit. For the low order digit, if the M flag in AC+3 (bit 2) is a 1, obtain the left half-word (negative digit) and if the M flag is a 0, obtain the right half-word (Positive digit). Continue conversion by updating both the length count contained in AC+4 until the length count is exhausted. The byte pointer contained in AC+4 is left addressing the last digit

stored in the destination string (low order digit), the length count is zero, AC and AC+1 contain zero. Now skip the next instruction in the sequence and continue sequential operation from there.

EDIT OPERATIONS

The EDIT operator performs a MINI-PROGRAM called the Editing Pattern that directs the translation and subsequent movement of a byte string from an area of memory called the source to another area of memory called the destination. The source and destination byte strings are addressed by standard KL10 byte pointers. The format of the translation table of E1 is identical to that for MOVST, but here the particular arrangement of the editing pattern determines how the destination string will ultimately be formed.

MINI-PROGRAM OVERVIEW

For the following discussion refer to Figure 10 Edit Basic Block Diagram. At the heart of the Edit operation is a string of 9-bit bytes called the Editing Pattern. This pattern is actually performed sequentially one byte at a time by the Edit operator, in a fashion similar to instructions in a program; hence, the term "MINI-PROGRAM". There are nine pattern bytes available, each consisting of 9-bits. Three of the pattern bytes provide the ability to skip forward over a number of pattern bytes in the MINI-PROGRAM, these are SKIPM, SKIPN, and SKIPA. The first two skips are condi-

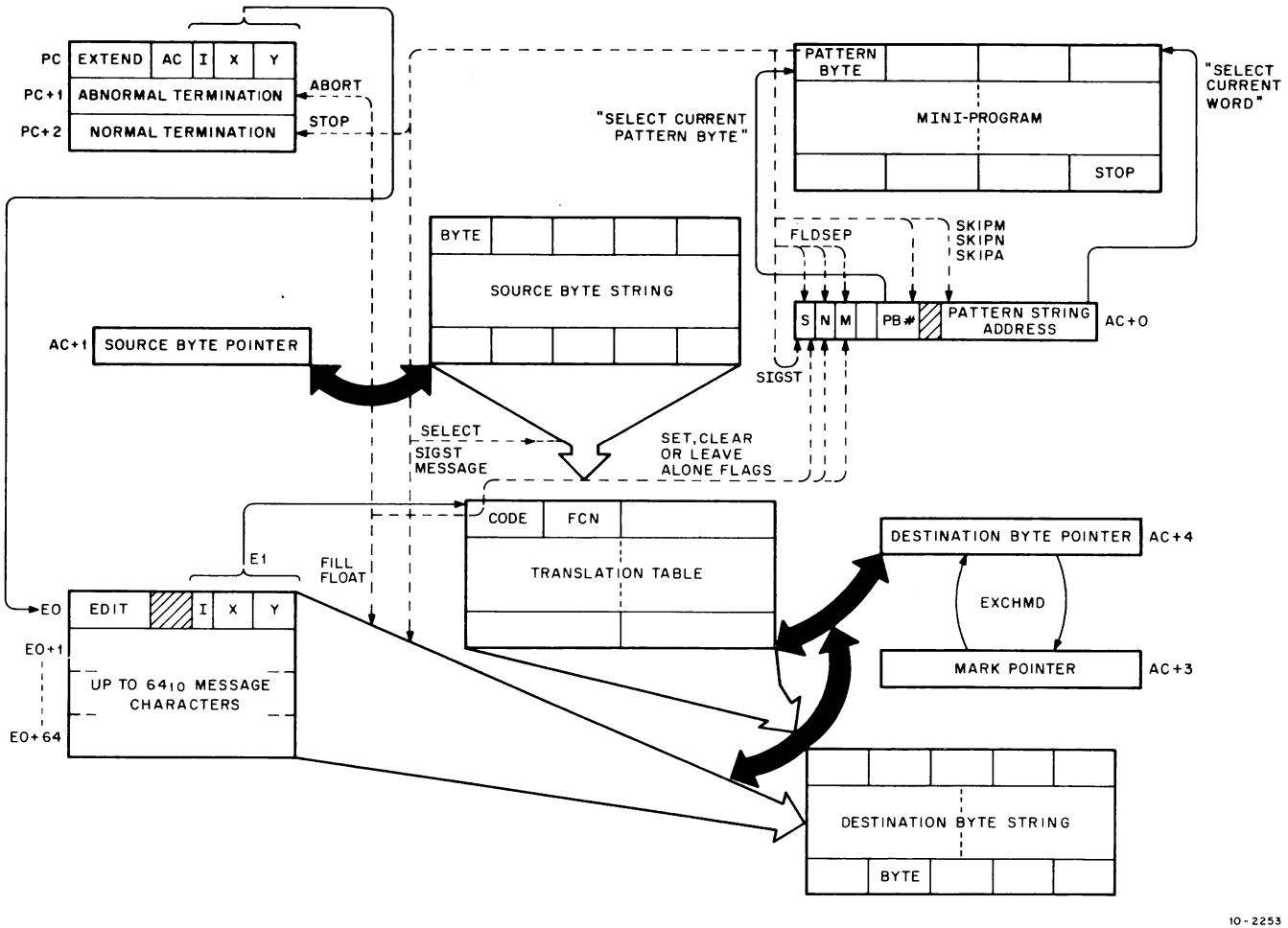
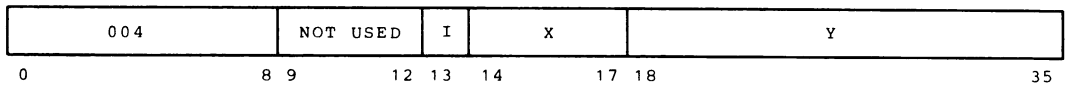


Figure 10 Edit Basic Block Diagram

tional on the state of the M and N flag respectively, while SKIPA is an unconditional skip. Another of the pattern bytes, STOP, provides for the normal termination of the editing operation. The pattern byte SELECT produces about the same function as the MOVST operator does for a single source byte. That is, it updates the source byte pointer, accesses the translation table to obtain the translation function, evaluates the code in the high order three bits, and then performs the indicated function. The pattern byte FLDSEP clears the S, N, and M flags in AC, while the EXCHMD pattern byte exchanges the contents of the MARK POINTER with that of the destination byte pointer. SIGST causes the S flag in AC to be set and then evaluates the contents of E0+2 to determine whether or not to store a float character in the destination string. Finally, MESSAG inserts a message character in the destination string. The MINI-PROGRAM is addressed by bits 4-5 (pattern byte number), and 18-35 (pattern string address) of AC. The programmer sets the initial value for these bits as desired, specifying one of four pattern bytes in bits 4-5 and specifying the particular word containing the pattern bytes in bits 18-35. The pattern byte number and pattern string address is updated during operation of the MINI-PROGRAM in one of two ways. After performing any one of the non-skip pattern bytes, the pattern byte number is incremented Modulo 4. Each time the pattern byte number becomes or is set to a value of three and is subsequently incremented causing it to become zero, the pattern string address is then incremented by +1

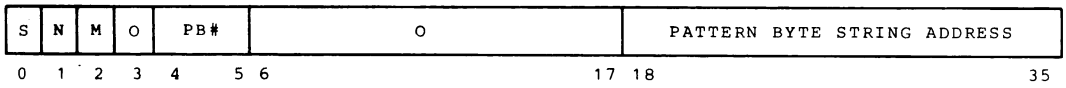
to address the next sequential word. When a skip pattern byte is met, the pattern byte number, as well as pattern string address, is incremented by the value specified in the skip pattern byte. Since there is no source length count associated with the Edit operator, termination of the editing operation is accomplished in one of two ways. The first occurs when a STOP pattern byte is performed and yields a SKIP (normal) return relative to the current PC. The second occurs when the ABORT-CODE is detected in the translation table and this yields the non-skip (abnormal) return once again relative to the current PC.

TECHNICAL SUMMARY (EDIT)



The effective address E1, which is calculated from I, X, and Y, is the base address of the translation table.

The following AC's are utilized by the EDIT operator.



Bits 0 - 2 of AC constitute the EDIT Flags. The meaning of each bit is as follows:

S FLAG - Significance. In EDIT the source string is scanned from left to right and each character is translated thru a table whose base address is E1. Three bits are associated with each half word entry in this table. The most significant bit of each of these entries is the S bit; when it is set in the table half word, it indicates that the character being translated is significant. When the S bit in the table is clear, reference is made to the fill character in location E0+1. The S flag in AC may, however, be set by the Programmer Explicitly, when the S bit in the translation table is set, or by performing the pattern byte operator SIGST. The influence of the S flag in AC, together with the S bit in the table, may be viewed as follows:

AC S FLAG	TABLE S BIT	IMPLICATION
0	0	NO SIGNIFICANCE. Fetch the contents of Location E0+1, and test it for equal to zero. If the contents of location E0+1 is zero, nothing is stored in the desti-

AC S FLAG	TABLE S BIT	IMPLICATION
		nation string. If the contents of location E0+1 is Non-Zero, store the contents of E0+1 (the fill character), in the destination string as specified by the destination byte pointer contained in AC+4.
0	1	SIGNIFICANT CHARACTER BEING TRANSLATED. Set both the S and the N Flags in AC. Store the destination byte pointer in the location specified by bits 18-35 of the MARK POINTER, which is AC+3. Now fetch the contents of location E0+2 (the float character), and test it for equal to zero. If the contents of location E0+2 is zero, store only the translated character in the destination string. If the contents of location E0+2 is non-zero, store the float character obtained from location E0+2 in the location specified by the destination byte pointer. Then update this pointer and use it once again to store the translated character. This effectively places the float character in front of the translated character in the destination string.

AC S FLAG	TABLE S BIT	IMPLICATION
S FLAG SETS	DON'T CARE	SIGST PATTERN OPERATOR BEING PERFORMED. Set the S Flag in AC and store the destination byte pointer in the location specified by bits 18-35 of the MARK POINTER, which is AC+3. Now fetch the contents of location E0+2 (the float character), and test it for equal to zero. If the contents of location E0+2 is zero, nothing is stored in the destination string. If the contents of location E0+2 is Non-Zero, store the float character obtained from location E0+2 in the location specified by the destination byte pointer.
1	X	S FLAG IN AC IS EQUAL TO A 1. The character being processed is treated as significant, and placed in the destination string. On performing the pattern operator message, if the S Flag in AC is set, the specified message character is fetched and placed in the destination string. Note that when the S flag in AC and the S

AC S FLAG	TABLE S BIT	IMPLICATION
		bit in the translation table are both zero, during a MESSAG pattern operation, that the contents of location E0+1 is fetched and tested for equal to zero. If the contents of location E0+1 is equal to zero, nothing is stored in the destination string. If the contents of location E0+1 is non-zero, the fill character is obtained thru location E0+1 and stored in the destination string.

N FLAG - NON-ZERO FLAG. This flag is set by the instruction if the significance starter bit is on in a table half-word (Code 4, 6, or 7). E.G., Code 4 sets both S and N in AC, Code 6 clears M and sets both S and N in AC, and Code 7 sets M, S, and N in AC. The Programmer must clear the N flag if this is desired since the instruction cannot provide that function.

M FLAG - MINUS FLAG. This flag may be set or cleared by the Programmer and may be set, cleared, or left alone by the sign control bits (CODES 2, 3, 6, and 7) in the translate table.

The following are the EDIT TABLE CONTROL CODES contained in bits 0-2 and 18-20 of the halfword translation functions.

CODE IN HIGH ORDER 3-BITS	FUNCTION PRODUCED
0	NOP. No effect on flags, and no significance indicated.
1	ABORT. Do not update the destination byte pointer contained in AC+4, and do not reference the destination string. Take the next instruction in the sequence and continue sequential operation from there.
2	CLEAR M. Clear the M Flag in AC by making it equal to 0.
3	SET M. Set the M Flag in AC to a 1.
4	SIGNIFICANCE START. The character being translated is significant. Set both the S Flag and the N Flag in AC to a 1.

CODE IN HIGH ORDER 3-BITS	FUNCTION PRODUCED
5	SIGNIFICANCE START AND ABORT. The character being translated is significant but do not reference the destination string. Set the N flag in AC to a 1. Do not update the destination byte pointer contained in AC+4. Take the next instruction in the sequence and continue sequential operation from there.
6	Set both the S Flag and the N Flag in AC to a 1, and clear the M Flag in AC.
7	Set the S Flag, the N Flag, and the M Flag to a 1.

Bits 4-5 of AC are used to select a 9 bit pattern byte operator from a word addressed by AC bits 18-35. The format of the pattern byte operator word is illustrated below in Figure 11.

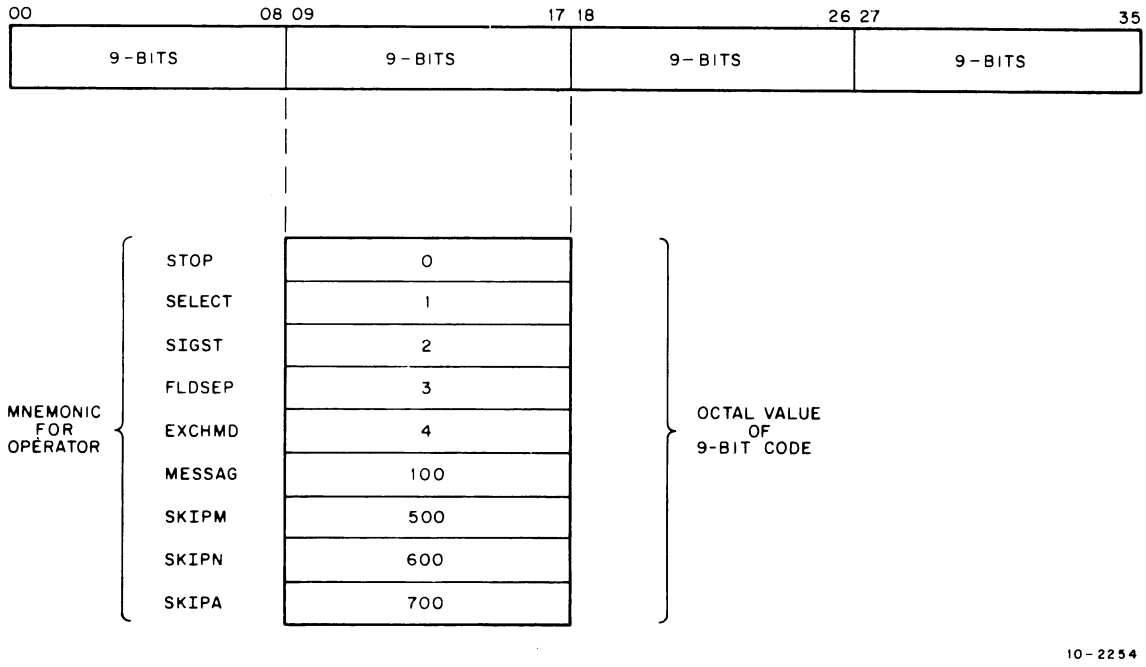


Figure 11 Pattern Byte Operator Word Format

The Edit Operator begins by using the specified contents of bits 4-5 of AC as its initial pattern byte selection. It then updates this to address the next sequential pattern byte. When the instruction reaches the fourth pattern byte in a pattern byte word it increments the word address (bits 18-35) in AC and sets the PB# to zero pointing to the left most pattern byte in the word.

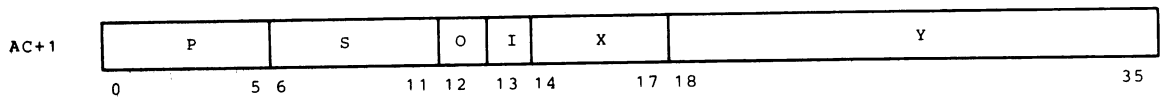
The following is a list of the specific pattern bytes and a description of their function.

PATTERN BYTE TYPE	OCTAL	FUNCTION
STOP	0	Terminate Edit. Terminate the editing operation. This is a normal Termination. Take the skip, return, (PC+2) and continue sequential operation from there.
SELECT	1	Select Next Source Byte. Update the source byte pointer and then obtain the specified source byte. Using the byte obtained, select the left or right halfword translation function from the table at E1. Now evaluate the code in the three high order bits of this entry together with the most significant bit of AC (S Flag), and perform the indicated function. For a detailed description of this function, see the Edit Technical Summary.

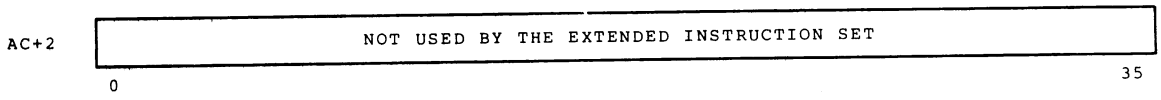
PATTERN BYTE TYPE	OCTAL	FUNCTION
SIGST	2	Significance Start. This pattern byte causes the S Flag in AC to be set indicating the beginning of significance in the source string. If the S Flag in AC was a zero, a copy of the destination byte pointer is placed in the location addressed by the mark pointer (AC+3). The contents of location E0+2 (the float character) is fetched and evaluated. If the contents of E0+2 is non-zero, then the destination byte pointer is updated and the float character is stored in the specified byte of the destination string. If the contents of E0+2 is equal to zero, the destination byte pointer is not updated and nothing is stored in the destination string.
FLDSEP	3	Field Separation. Clear the S, N and M Flags in AC.
EXCHMD	4	Exchange Mark and Destination Pointers. Exchange the contents of the location

PATTERN BYTE TYPE	OCTAL	FUNCTION
EXCHMD (Cont.)	4	addressed by bits 18-35 of AC+3 (the mark pointer) with the contents of AC+4 (the destination byte pointer).
MESSAG	100	Insert Message Character. Test the S Flag in AC. If the S Flag is equal to 0, update the destination byte pointer and store the contents of E0+1 (the fill character) in the specified byte of the destination string. If the S Flag is equal to 1, update the destination byte pointer and store the indicated message character in the specified byte of the destination string. The format of the MESSAG pattern byte includes a provision for addressing up to 64 message characters. These message characters may be contained in locations E0+1 thru E0+64 where the first such character is the fill character (E0+1)aand the second is the float character (E0+2).

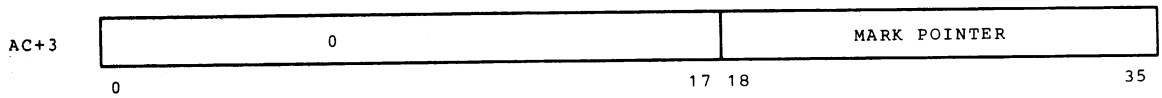
PATTERN BYTE TYPE	OCTAL	FUNCTION
SKIPM	500	Skip the next n+1 pattern bytes if the M Flag in AC is equal to 1. All skip pattern bytes includes in their format a provision for specifying the number of pattern bytes to skip. The range of the skip is 1-64. Thus for skip M+0, the next sequential pattern byte will be performed only if the M Flag is equal to 0, otherwise the next sequential pattern byte is skipped.
SKIPN	600	Skip the next n+1 pattern bytes if the N Flag in AC is equal to 1. Same format as SKIP M, but the skip is conditional on the state of the N Flag in AC.
SKIPA	700	Unconditionally. Skip the next n+1 pattern bytes.



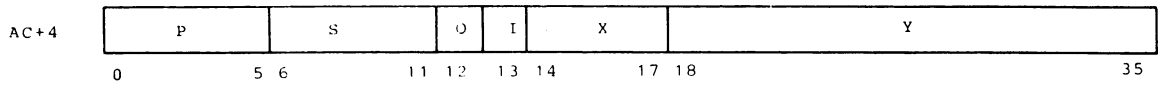
This is a standard KL10 byte pointer, and addresses the sources byte string. See Section 2.3 for a description of byte manipulation.



This accumulator is not disturbed when the operator is performed.



AC+3 contains the address where AC+4 (the destination byte pointer) is to be stored when significance in the source string is detected.



This is a standard KL10 byte pointer, and addresses the destination byte string. See Section 2.3 for a description of byte manipulation.

LOGICAL FLOW (EDIT)

Process the source byte string specified by the byte pointer contained in AC+1 under control of the EDITING PATTERN addressed by bits 4-5 and 13-35 of AC. Bits 13-35 address a block of words, where each word contains up to four 9-bit pattern byte operators, and bits 4-5 selected the initial pattern byte operator to be used.

Replace the destination string specified by the byte pointer contained in AC+4 with the edited result. Proceed one byte at a time, beginning with the specified pattern operator and continue until either a STOP pattern operator is encountered or an ABORT CODE is detected in the translation table. Perform the specified editing utilizing the translation table addressed by E1 and up to 64 message characters which may be contained in locations E0+1 thru E0+64.