digital

# decsystem10 assembly language handbook

third edition

system reference
macro
monitor calls

link-10
ddt
utilities

decsystem10 handbook series

digital

# decsystem10 assembly language handbook

third edition

Additional copies of this handbook may be ordered from:
Software Distribution Center, DEC, Maynard, Mass. 01754. Order Code: DEC-10-NRZC-D.

decsystem10 handbook series

| | |
|---|---|
| DEC | PDP |
| FLIP CHIP | FOCAL |
| DIGITAL | COMPUTER LAB |

# tab index

# FOREWORD

This handbook is a collection of documents and sections of documents taken from the DECsystem-10 SOFTWARE NOTEBOOKS (DEC-10-SYZB-D). It is intended to be used by experienced programmers interested in writing and operating assembly-language programs on the DECsystem-10. The material in this handbook is aimed at providing the information needed for user-mode programming.

Most documents in this handbook are reprinted without change from the DECsystem-10 Software Notebooks. However, the first document in the handbook, the System Reference Manual, is an excerpt from the System Reference Manual in the Notebook set. This excerpt contains only the user-mode programming information needed by most assembly language programmers, and does not cover the documentation related to the various peripheral devices. If additional information is required, the reader is referred to the complete System Reference Manual in the Software Notebooks. All DECsystem-10 installations have two copies of the notebook set for reference.

The documents contained in this handbook reflect the following hardware and versions of the software:

System Reference Manual – KA10 and KI10 processors
MACRO – Version 47
Monitor Calls – 5.06 release
DDT – Version 34
LINK-10 – Version 1
CREF – Version 47
FILCOM – Version 20
FUDGE2 – Version 15
GLOB – Version 5A

The Assembly Language Handbook is one in the set of DECsystem-10 handbooks. The other handbooks comprising this series are:

(1) the COBOL Language Handbook.

(2) the Mathematical Languages Handbook, which covers FORTRAN, BASIC and ALGOL.

(3) the DECsystem-10 Users Handbook, which includes an introductory section, the operating system commands, TECO, and PIP.

In addition to the above-mentioned handbooks, the following documentation is also available:

(1) the COBOL Users Guide, which is aimed at COBOL users who wish to become familiar with COBOL on the DECsystem-10.

(2) the System Reference Card, which includes the word formats, instructions, and conversion tables for the DECsystem-10.

(3) the Operating System Commands Reference Card, which describes the commands, along with their formats, that are a part of the operating system.

(4) the Monitor Calls Reference Card, which covers the programmed operators (UUOs), and their formats, that can be used with the monitor.

(5) the BASIC Language Reference Card, which includes the statements, intrinsic functions, and edit and control commands of the DECsystem-10 BASIC Language.

The handbooks, Users Guide, and reference cards may be ordered from:

Software Distribution Center
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

# DECsystem-10

# System Reference Manual

ORDER NO. DEC-10-HGAD-D FROM PROGRAM LIBRARY, MAYNARD, MASSACHUSETTS PRICE $5.00

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.

December 1971

First edition, May 1968
Three printings

Second edition, December 1971

This edition has been expanded to provide system reference information for a DECsystem–10 with KA10 or KI10 central processors. The KI10 material has been incorporated into the text throughout.

Manufactured in the United States of America

# Contents

# 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).

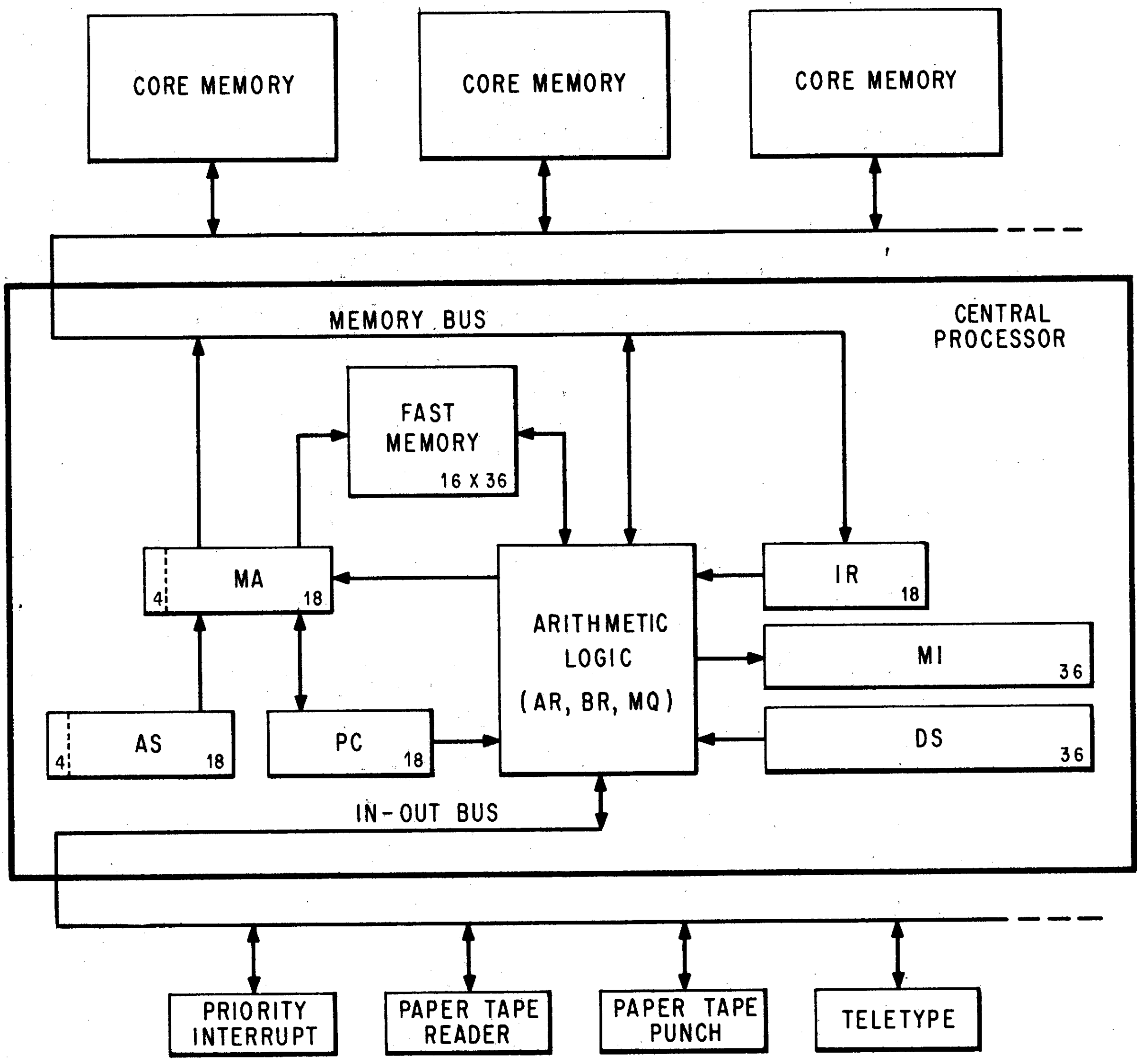
Confusion could result only in a chapter dealing with a small-computer subsystem. Here the small processor is usually referred to by its name (PDP-8, PDP-11) and the words "computer" and "memory" refer to the small computer. To differentiate, the PDP-10 is referred to by its name or as the "DECsystem-10 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 two types of PDP-10 central processors, the KA10 and the KI10. The latter is faster and more powerful, having a somewhat larger instruction repertoire including double precision floating point. Both 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 both processors use 18-bit addresses and can thus reference 262,144 word locations in memory. This is the total addressing capability of the KA10, but in the KI10 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. KI10 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 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. By means of the address switch register AS, the operator can 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. The instruction register IR contains the left half of the current instruction word, *ie* all but the address part. The memory address register MA 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 BR 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. In the KI10, AR and the adder each have a 28-bit left extension for handling double precision floating point numbers.

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

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, 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 allows unrestricted in-out with a limited number of devices for special real time applications.

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

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.

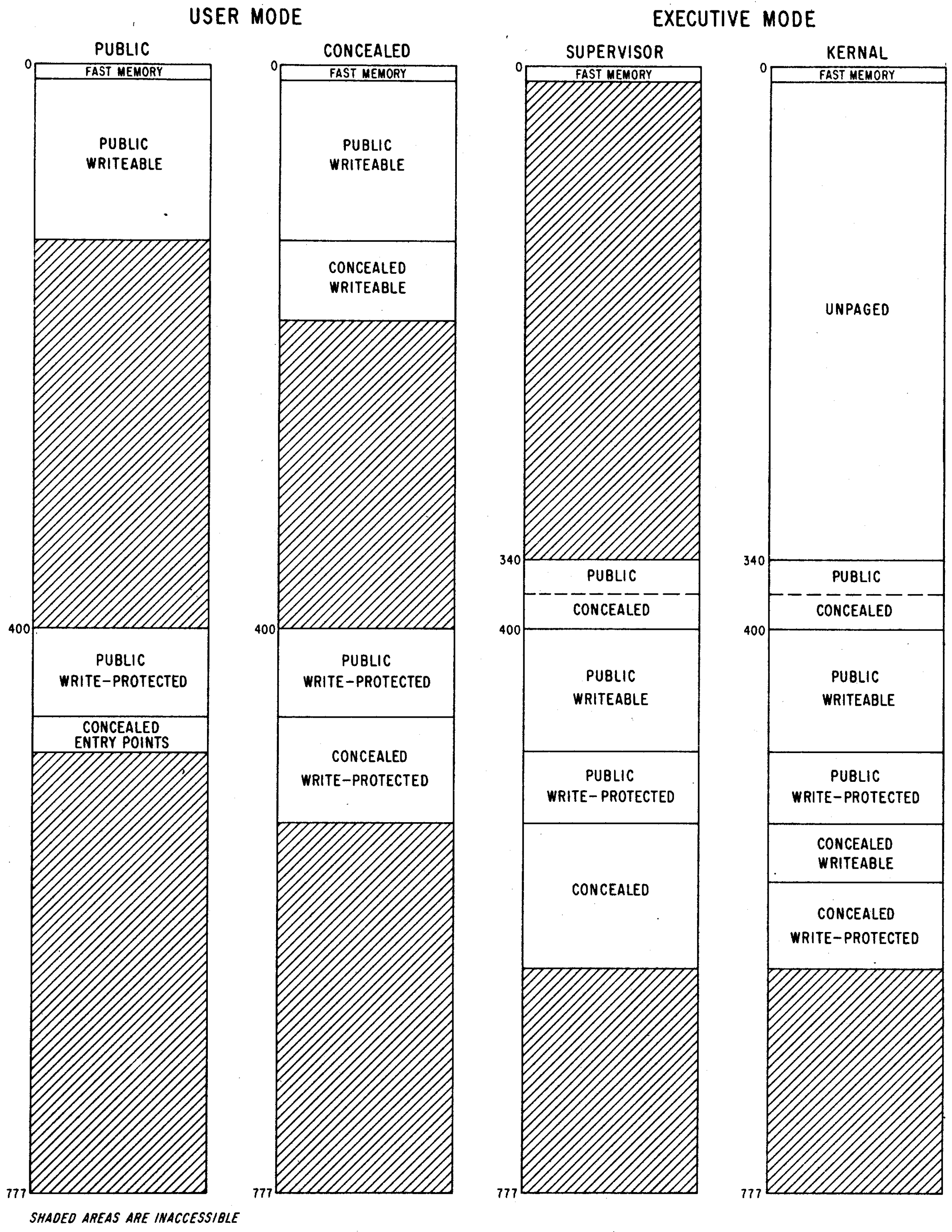
The KI10 has paging hardware for the mapping of pages from the limited virtual address space into pages anywhere in physical memory. A page map for each program specifies not only the correspondence from virtual address to physical address, but also whether an individual page is accessible or not, alterable 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 the program can even address some of memory directly (*ie* unpaged); 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.

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



TYPICAL VIRTUAL ADDRESS SPACE CONFIGURATION 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.

## 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 as 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.

$$+153_{10} = +231_8 = \underset{0\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 35}{\boxed{000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 010\ 011\ 001}}$$

$$-153_{10} = -231_8 = \underset{0\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 35}{\boxed{111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 101\ 100\ 111}}$$

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.

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 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.

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 0. The range for double length integers and proper fractions is thus $-2^{70}$ to $2^{70} - 1$ and $-1$ to $1 - 2^{-70}$.

**Floating Point Arithmetic.** The KI10 has 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.

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

| 0 | 10 001 000 | 100 110 010 000 000 000 000 000 000 |
|---|---|---|
| 0 | 1 8 | 9 35 |

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

| 1 | 01 110 111 | 011 001 110 000 000 000 000 000 000 |
|---|---|---|
| 0 | 1 8 | 9 35 |

. 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 double precision format:

```
|0|10 010 011|100 000 000 000 000 000 000 000 000|
 0 1        8 9                                35

|0|01 111 000|000 000 000 100 000 000 000 000 000|
 0 1        8 9                                35
```

whereas its negative is

```
|1|01 101 100|011 111 111 111 111 111 111 111 111|
 0 1        8 9                                35

|0|01 111 000|111 111 111 100 000 000 000 000 000|
 0 1        8 9                                35
```

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:

```
|0|10 010 011|100 000 000 000 000 000 000 000 000|
 0 1        8 9                                35

|0|00 000 000 010 000 000 000 000 000 000 000 000|
 0 1                                           35
```

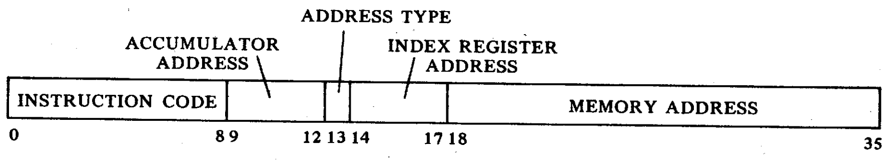
and its negative is

```
|1|01 101 100|011 111 111 111 111 111 111 111 111|
 0 1        8 9                                35

|0|11 111 111 110 000 000 000 000 000 000 000 000|
 0 1                                           35
```

## 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

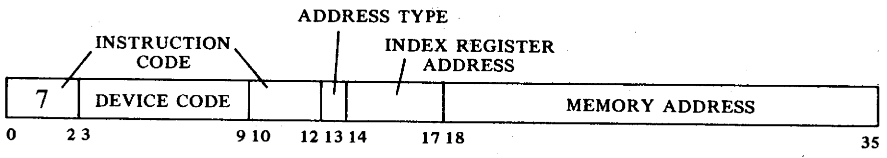


BASIC INSTRUCTION FORMAT

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

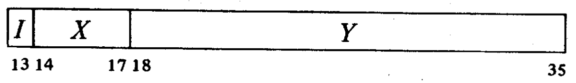
An input-output instruction is designated by three 1s in bits 0–2. Bits 3–9 address the in-out device to be used in executing the instruction, and bits 10–12 specify the operation. The rest of the word is the same as in other instructions.

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 G2*].

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.

## 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.

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 a word directly, but to write it 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 usually considerably shorter. The size of the MD10 can be increased in units of 32K up to 128K.

| | *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 }<br>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 |
| KA10 Fast Memory | .21 | .21 | | | 16 |
| KI10 Fast Memory | | | | | 16 |

*Add .1 in a multiprocessor system.

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 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 in 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 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).

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 almost all of these are relative to pages whose physical location is specified by the Monitor. The only 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) |

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.

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.

## 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
```

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.

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

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 tc 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.

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 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, §2.13 describes the priority interrupt, §2.14 discusses trapping, and §§2.15 and 2.16 describe the special effects and restrictions associated with the various machine modes in the KI10 and the KA10 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**

| 500 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| 510 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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

| 520 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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

| 530 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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

| | | | | | |
|---|---|---|---|---|---|
| 504 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 | 17 18 35 |

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

| | | | | | |
|---|---|---|---|---|---|
| 514 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 | 17 18 35 |

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

| | | | | | |
|---|---|---|---|---|---|
| 524 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 | 17 18 35 |

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**

| 534 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 540 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 550 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 560 | *M* | *A* | *I* | *X* | *Y* |

0 6 7 8 9 12 13 14 17 18 35

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**

| | | | | | |
|---|---|---|---|---|---|
| 570 | *M* | *A* | *I* | *X* | *Y* |

0 6 7 8 9 12 13 14 17 18 35

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**

| | | | | | |
|---|---|---|---|---|---|
| 544 | *M* | *A* | *I* | *X* | *Y* |

0 6 7 8 9 12 13 14 17 18 35

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.

HLRI merely clears AC right.

| | | |
|---|---|---|
| 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 |

### HLRZ Half Word Left to Right, Zeros

| | | | | | |
|---|---|---|---|---|---|
| 554 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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.

HLRZI merely clears AC and is thus equivalent to HLLZI.

| | | |
|---|---|---|
| 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 |

### HLRO Half Word Left to Right, Ones

| | | | | | |
|---|---|---|---|---|---|
| 564 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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.

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

| | | |
|---|---|---|
| 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 |

### HLRE Half Word Left to Right, Extend

| | | | | | |
|---|---|---|---|---|---|
| 574 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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
```

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
```

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.)

## 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

| 0–8 | 9–12 | 13 | 14–17 | 18–35 |
|---|---|---|---|---|
| 250 | *A* | *I* | *X* | *Y* |

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

**BLT** **Block Transfer**

| 251 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 — 8 | 9 — 12 | 13 | 14 — 17 | 18 — 35 |

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 $E - \mathrm{AC_R} + 1$.

*Caution*

Priority interrupts are allowed during the execution of this instruction, following the processing of each word. If an interrupt 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, *A* and *X* must not address the same register as this would produce a different effective address calculation upon resumption should an interrupt occur; and the program 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.

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**

| | | | | | |
|---|---|---|---|---|---|
| 200 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 204 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 210 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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 $-1 \times 2^{127}$ 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.

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

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

**MOVM Move Magnitude**

| 214 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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 $-1 \times 2^{127}$ 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.

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

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

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

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**

| 261 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 $1\,000001_8$ to the entire register. In the KI10 the two halves are handled independently.

**POP** **Pop Up**

| 262 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

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.

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.

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 *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)
```

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

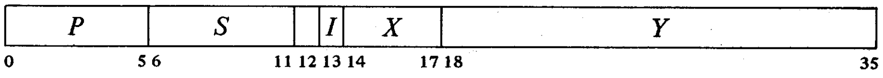
removes the last item and inserts it in place of the next to last item in the shortened list.

## 2.3 BYTE MANIPULATION

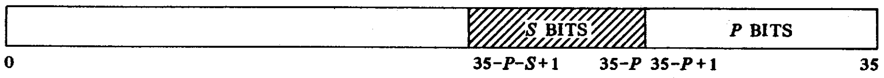
In a KA10 without byte manipulation hardware, all of the instructions presented in this section are trapped as unassigned codes [§2.10].

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



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. 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 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**

| 135 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 137 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 133 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

Increment the byte pointer in location $E$ as explained above.

**ILDB** **Increment Pointer and Load Byte**

| 134 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 136 | A | I | X | Y |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 $P$.

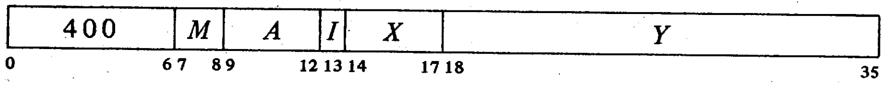
## 2.4 LOGIC

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**

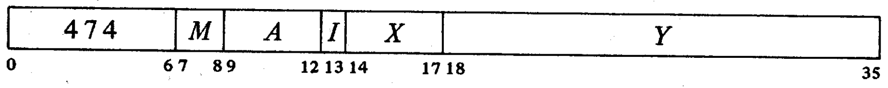


Change the contents of the destination specified by *M* to all 0s.

SETZ and SETZI are equivalent (both merely clear AC). 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**

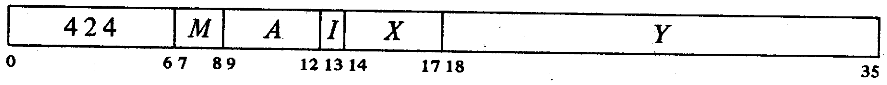


Change the contents of the destination specified by *M* to all 1s.

SETO and SETOI are equivalent.

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

**SETCA  Set to Complement of AC**

| 450 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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).

**SETM  Set to Memory**

| 414 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

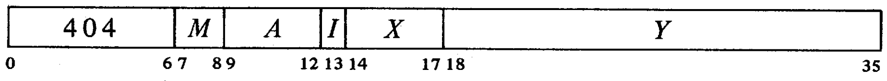
| 460 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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

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

| | | |
|---|---|---|
| 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 |

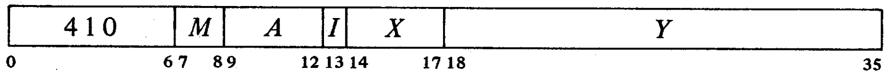
**AND And with AC**



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 |

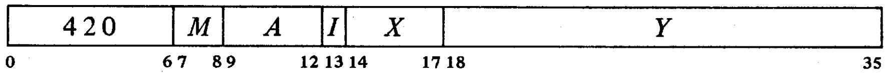
**ANDCA And with Complement of AC**



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**

| | | | | | |
|---|---|---|---|---|---|
| 440 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 434 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 454 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 464 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 470 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 430 | *M* | *A* | *I* | *X* | *Y* |
| 0 | 6 7 | 8 9 | 12 13 | 14 17 | 18 35 |

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**

| 444 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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 |

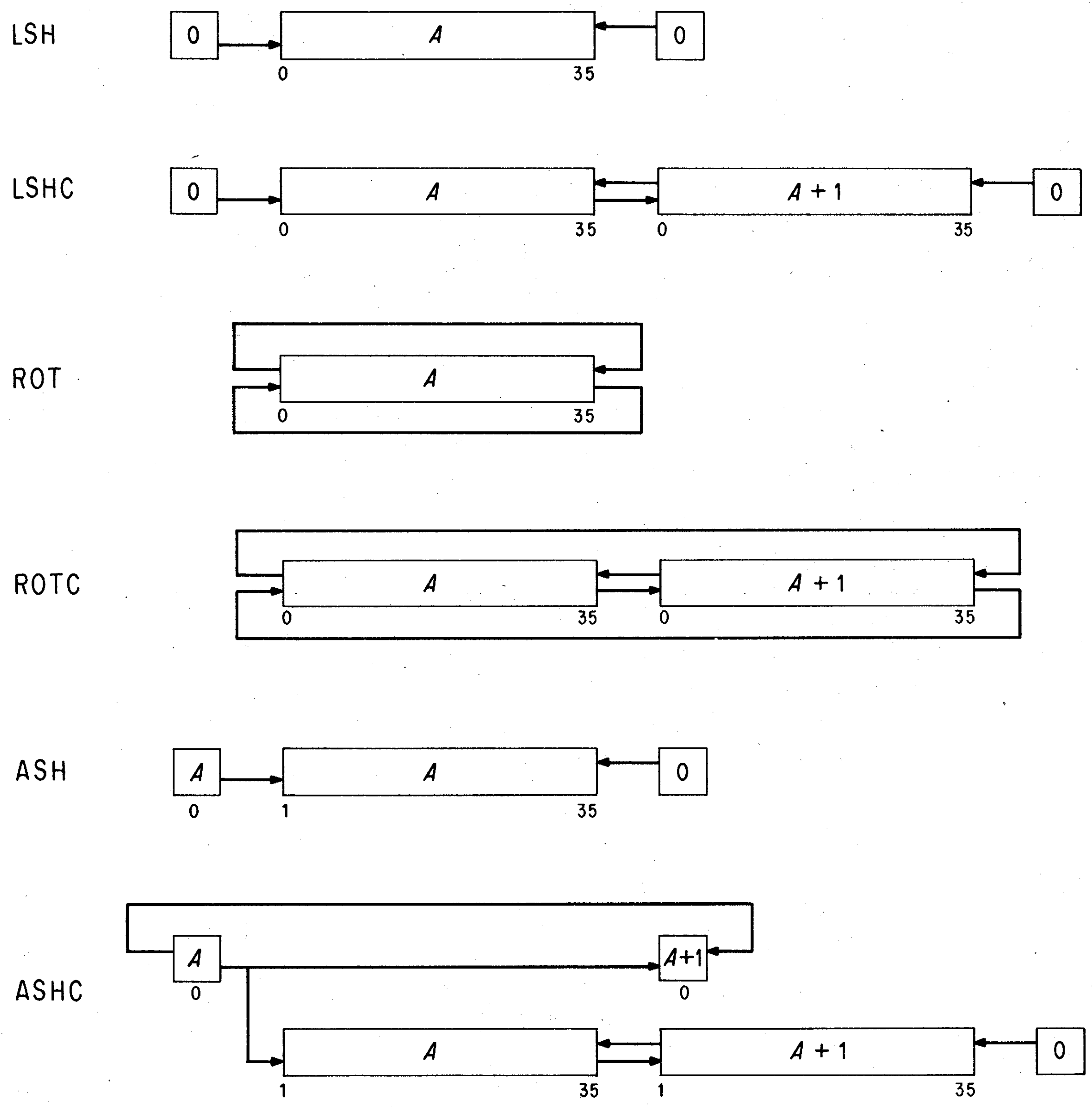
The original contents of the destination can be recovered except in EQVB, where both operands are replaced by the result. In the other three modes the replaced operand is restored by repeating the instruction in the same mode, *ie* by taking the equivalence function of the remaining operand and the result.

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 of the operand by shifting $E$ mod 72 places, for a maximum of 71.

**LSH Logical Shift**

| | | | | |
|---|---|---|---|---|
| 242 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | |
|---|---|---|---|---|
| 246 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | |
|---|---|---|---|---|
| 241 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 245 | $A$ | $I$ | $X$ | $Y$ |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 only single length numbers, whereas multiply supplies a double length product, and divide uses a double 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. 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 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.

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

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]. In the KA10, Overflow is available as a processor condition (via an in-out instruction) that can request a priority interrupt if enabled, whereas KI10 overflow is handled by trapping through the setting of Trap 1 [*both subjects are discussed in* §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.

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**

| | | | | | |
|---|---|---|---|---|---|
| 270 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

Add the operand specified by *M* to AC and place the result in the specified destination. If the sum is $\geqslant 2^{35}$ 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**

| | | | | | |
|---|---|---|---|---|---|
| 274 | *M* | *A* | *I* | *X* | *Y* |
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

Subtract the operand specified by *M* from AC and place the result in the specified destination. If the difference is $\geqslant 2^{35}$ 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**

| 224 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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 $-2^{70}$.

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

**IMUL** **Integer Multiply**

| 220 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 234 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

If the magnitude of the number in AC is greater than or equal to that 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.

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

**IDIV** **Integer Divide**

| 230 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

If the operand specified by *M* is zero, 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 |

Example. 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.

As an example suppose we wish to determine the parity of the 8-bit character *abcdefgh*, where the letters represent the bits of the character. Assuming the character is right-justified in AC, we first 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 (with the character right-justified in AC):

```
          IMULI   AC,200401
          AND     AC,ONES
          IMUL    AC,ONES
            .
            .
            .
ONES:     11111111
```

where the parity is indicated by AC bit 14. Of course, following the IMUL would be a test instruction to check the value of the bit.

### 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 of the operand by shifting $E$ mod 72 places, for a maximum of 71.

**ASH** **Arithmetic Shift**

| 240 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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
```

This takes 5–6 μs as opposed to about 16 μs for IDIVI.

**ASHC** **Arithmetic Shift Combined**

| 244 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

## 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]. In the KA10, Overflow and Floating Overflow are available as processor conditions (via an in-out instruction) that can request a priority interrupt if enabled, whereas KI10 overflow is handled by trapping through the setting of Trap 1 [*both of these subjects are discussed in* §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 $0 \times 2^{69}$ 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**

| 132 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 has 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**

| 122 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

This overflow test checks for a value $\geqslant 2^{35}$ 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**

| 126 | A | I | X | Y |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

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

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.

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**

| 127 | A | I | X | Y |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 144 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 154 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 164 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 174 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

Usually the double length number is in two adjacent accumulators, and $E$ equals $A$+1.

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.

**DFN** **Double Floating Negate**

| | | | | |
|---|---|---|---|---|
| 131 | $A$ | $I$ | $X$ | $Y$ |
| 0 … 8 | 9 … 12 | 13 | 14 … 17 | 18 … 35 |

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.

**UFA** **Unnormalized Floating Add**

| | | | | |
|---|---|---|---|---|
| 130 | $A$ | $I$ | $X$ | $Y$ |
| 0 … 8 | 9 … 12 | 13 | 14 … 17 | 18 … 35 |

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 $\geqslant 1$, and place the high order part of the result in AC $A$+1. The original contents of AC and $E$ are unaffected.

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.

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.

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.

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 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*. |

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**

| 140 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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 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 $> 154\ (127+27)$ 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 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**

| 150 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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 $> 154\ (127+27)$ 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 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**

| 160 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 170 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 6 | 7 8 | 9 12 | 13 | 14 17 | 18 35 |

If the magnitude of the fraction in AC is greater than or equal to twice that 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 receives rubbish.

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 $> 127$ or $< -128$ 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. 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 all cases the result, rounded except in 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**

| | | | | |
|---|---|---|---|---|
| 110 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

Floating add the operand of locations *E* and *E*+1 to the operand of accumulators *A* and *A*+1. If the high order 70 bits of the fraction in the sum are 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

| 111 | $A$ | $I$ | $X$ | $Y$ |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

Floating subtract the operand of locations $E$ and $E+1$ from the operand of accumulators $A$ and $A+1$. If the high order 70 bits of the fraction in the difference are 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

| 112 | $A$ | $I$ | $X$ | $Y$ |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

Floating multiply the operand of accumulators $A$ and $A+1$ by the operand of locations $E$ and $E+1$. If the high order 70 bits of the fraction in the product are zero, clear $A$ and $A+1$. Otherwise, if there are any bits of significance among the high order 35, do at most one normalization shift if required; if the high order 35 bits are zero, shift the fraction left 35 places (adjusting the exponent), and then do at most one normalization shift if required. Round the high order double length part, test for exponent overflow and underflow as described above, and place the result in ACs $A$ and $A+1$.

The 35-bit shift can be done only if the original operands are unnormalized.

### DFDV Double Floating Divide

| 113 | $A$ | $I$ | $X$ | $Y$ |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 of 62 bits. If the fraction

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

is zero, clear *A* and *A*+1. Otherwise 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**

| | | | | |
|---|---|---|---|---|
| 120 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | |
|---|---|---|---|---|
| 124 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

Do not use the instruction DMOVEM AC,AC+1. At present the processor places AC in both AC+1 and AC+2, but this result is not guaranteed.

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.

**DMOVN Double Move Negative**

| | | | | |
|---|---|---|---|---|
| 121 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

Note that these two instructions can be used to negate numbers in hardware double precision format only; for software double precision, the program must use DFN.

Note also that there is no overflow test, as negating a correctly formatted floating point number cannot cause overflow.

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.

**DMOVNM Double Move Negative to Memory**

| | | | | |
|---|---|---|---|---|
| 125 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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

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.

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.

```
SETCM   AC,E          ;Take ones complement of high word
MOVN    AC+1,E+1      ;Take twos complement of low word
TDNN    AC+1,[377777777777]   ;If low part of fraction is
ADDI    AC,1          ;zero, change high word to twos com-
                      ;plement
```

sent the processor 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.

## 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

| | | | | |
|---|---|---|---|---|
| 252 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| | | | | |
|---|---|---|---|---|
| 253 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 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 any condition that sets Overflow also sets the Trap 1 flag.

**CAI Compare AC Immediate and Skip if Condition Satisfied**

| 30 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

**CAM Compare AC with Memory and Skip if Condition Satisfied**

| 31 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 32 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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

JUMP is a no-op (instruction code 320 has this mnemonic for symmetry).

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

| | | |
|---|---|---|
| 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**

| 33 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

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

| | | |
|---|---|---|
| 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 |

**AOJ Add One to AC and Jump if Condition Satisfied**

| 34 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 35 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 36 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 37 | *M* | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|
| 0 5 | 6 8 | 9 12 | 13 | 14 17 | 18 35 |

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
```

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

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

## 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**

| 60 | *M* | 0 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

| | | |
|---|---|---|
| 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**

| 62 | *M* | 0 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | | |
|---|---|---|---|---|---|---|
| 64 | *M* | 0 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | | |
|---|---|---|---|---|---|---|
| 66 | *M* | 0 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | | |
|---|---|---|---|---|---|---|
| 60 | *M* | 1 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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 | TLN is a no-op. |
| 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 | |

**TLZ** **Test Left, Zeros, and Skip if Condition Satisfied**

| 62 | *M* | 1 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 64 | *M* | 1 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 66 | *M* | 1 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 61 | *M* | 0 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 63 | *M* | 0 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 65 | *M* | 0 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| 67 | *M* | 0 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| 61 | *M* | 1 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| 63 | *M* | 1 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|---|---|
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | | |
|---|---|---|---|---|---|---|
| 65 | *M* | 1 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | | |
|---|---|---|---|---|---|---|
| 67 | *M* | 1 | *A* | *I* | *X* | *Y* |
| 0 5 | 6 7 | 8 | 9 12 | 13 | 14 17 | 18 35 |

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:

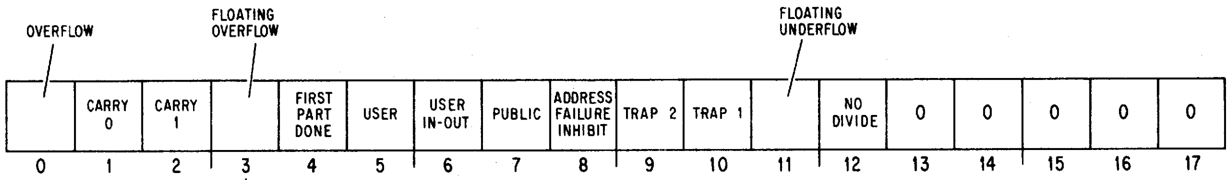
```
TRC     F,X+Y      ;Complement flags X and Y
TRCE    F,X+Y      ;Test both and restore original states
. . .              ;Do this if not both on
. . .              ;Skip to here if both on
```

## 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. Hence in these instructions one can store any information in bits 18–35 without fear of attempting to address a location outside a user block or in a memory that does not exist.

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:<br>A single instruction has set one of the carry flags (bits 1 and 2) without setting the other.<br>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.<br>An MUL*X* has multiplied $-2^{35}$ by itself (product $2^{70}$).<br>An IMUL*X* has multiplied two numbers with product $\geqslant 2^{35}$ or $< -2^{35}$.<br>An FIX or FIXR has fetched an operand with exponent $> 35$.<br>Floating Overflow has been set (bit 3).<br>No Divide has been set (bit 12). |
| 1 | Carry 0 – if set without Carry 1 (bit 2) being set, causes Overflow to be set and indicates that one of the following has occurred:<br>An ADD*X* has added two negative numbers with sum $< -2^{35}$.<br>An SUB*X* has subtracted a positive number from a negative number with difference $< -2^{35}$.<br>An SOJ*X* or SOS*X* has decremented $-2^{35}$.<br>But if set with Carry 1, indicates that one of these nonoverflow events has occurred:<br>In an ADD*X* both summands were negative, or their signs differed and their magnitudes were equal or the positive one was the greater in magnitude.<br>In an SUB*X* 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.<br>An AOJ*X* or AOS*X* has incremented $-1$.<br>An SOJ*X* or SOS*X* has decremented a nonzero number other than $-2^{35}$.<br>An MOVN*X* has negated zero. |

In user mode, bit 0 reflects the state of Overflow. But when the flags are saved in KI10 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.

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 without 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 $\geqslant 2^{35}$.

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

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

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

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:

In a floating point instruction other than FLTR, DMOVN, 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 DMOVMN, 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].

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

In the KA10, User In-out is applicable only to user mode [§2.16]. In the KI10 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].

7 Public (KI10 only) – 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 only) – an address failure cannot occur during the next instruction [§2.15].

9 Trap 2 (KI10 only) – if bit 10 is not also set, arithmetic overflow has occurred. If traps are enabled, the setting of this flag immediately causes one [§2.14]. At present, bits 9 and 10 cannot be set together by any hardware condition.

10 Trap 1 (KI10 only) – if bit 9 is not also set, pushdown overflow has occurred. If traps are enabled, the setting of this flag immediately causes one [§2.14]. At present, bits 9 and 10 cannot be set together by any hardware condition.

11 Floating Underflow – in a floating point instruction other than FLTR, DMOVN, DMOVNM or DFN, 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.

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

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

**XCT** **Execute**

| | | | | |
|---|---|---|---|---|
| 256 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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).

A user XCT or any KA10 XCT acts as described here, and the *A* portion of the instruction is ignored. But 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].

**JFFO** **Jump if Find First One**

| | | | | |
|---|---|---|---|---|
| 243 | *A* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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 an integer in AC:

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

**JFCL** **Jump on Flag and Clear**

| | | | | |
|---|---|---|---|---|
| 255 | *F* | *I* | *X* | *Y* |
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 264 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

The *A* portion of this instruction is ignored.

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.

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**

| 265 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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

| 254 | *F* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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]. Unless the User In-out flag is set, this function cannot be performed in a user program. Instead of restoring the channel, it acts just like an MUUO [§2.10]. |
| 10 | Halt the processor. When it stops, the MA lights on the console 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). Unless the User In-out flag is set, this function cannot be performed in a user program. 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 by the Monitor, *ie* if User is clear. A 1 in bit 7 sets Public, but a 0 |

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, MA points 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.

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.* 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.

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 |
| | 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.

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**

| 266 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 267 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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**

| 260 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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. 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 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**

| 263 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 8 | 9 12 | 13 | 14 17 | 18 35 |

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.

The effective address *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.

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,

## 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

| 001 – 037 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0 – 8 | 9 – 12 | 13 | 14 – 17 | 18 – 35 |

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

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.

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.

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.

The unassigned codes are 100–107, 114–117, 123 and 247.

**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, set up the flags and PC according to a new PC word taken from a third 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 or overflow).

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 if overflow traps are enabled, setting a trap flag immediately causes one.

**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, 247 and 257. 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.

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

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.

The instruction repertoires of the KA10, the KI10 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
JRST    KI10        ;KI10 if AC = 0 (no carry between
                    ;halves)
```

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.

In the standard algorithm for converting a number $N$ to its equivalent in base $b$, one performs the series of divisions

$$\begin{array}{rcll} N/b & = & q_1 + r_1/b & r_1 < b \\ q_1/b & = & q_2 + r_2/b & r_2 < b \\ q_2/b & = & q_3 + r_3/b & r_3 < b \\ \vdots & & & \\ q_{n-1}/b & = & 0 + r_n/b & r_n < b \end{array}$$

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:

$$\begin{array}{rcl} 61/8 & = & 7 + 5/8 \\ 7/8 & = & 0 + 7/8 \end{array}$$

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            ;12₈ = 10₁₀
        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 making the following substitutions.

$$\text{PUSH P,T+1} \rightarrow \text{HRLM T+1,(P)}$$
$$\text{POP P,T} \rightarrow \text{HLRZ T,(P)}$$

The routine can handle a 36-bit unsigned integer if the IDIVI T,12 is replaced by

```
        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
```

MACRO interprets a number following ↑D as decimal.

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 [§2.6 *describes the floating point instructions*].

```
DFAD:   UFA     A+1,M+1         ;Sum of low parts to A+2
```

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.

```
        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 algorithm

$$\frac{A}{B} \cong q + \frac{(r - qd) \times 2^{-27}}{b}$$

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,
```

## 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 instruction restrictions. In the KA10 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 kernel 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.

**CONO**　**Conditions Out**

| 7 | D | 20 | I | X | Y |
|---|---|---|---|---|---|
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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**

| 7 | D | 24 | I | X | Y |
|---|---|---|---|---|---|
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 7 | *D* | 14 | *I* | *X* | *Y* |
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 7 | *D* | 04 | *I* | *X* | *Y* |
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 7 | *D* | 30 | *I* | *X* | *Y* |
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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**

| | | | | | |
|---|---|---|---|---|---|
| 7 | *D* | 34 | *I* | *X* | *Y* |
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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**

| 7 | D | 10 | I | X | Y |
|---|---|---|---|---|---|
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

**BLKI** **Block In**

| 7 | D | 00 | I | X | Y |
|---|---|---|---|---|---|
| 0 2 | 3 9 | 10 12 | 13 | 14 17 | 18 35 |

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.

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.

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 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 C).

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 teletype 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 teletype 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 teletype output are like the reader. Teletype 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.

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, selects physical page 0 for the executive process table, and disables overflow traps. 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 memory (including fast memory) except in location 0. The operation affects none of memory except location 0 and the block area. For the KI10 it is recommended that read in be confined to the unpaged area, as bringing data into locations above 337777 would require prior loading of the appropriate pointers into the executive page map in physical page 0.

Upon completing the block, the processor halts only if the single instruction switch is on. Otherwise it leaves readin mode and begins normal operation. This is done in the KI10 by jumping to the location addressed by the last word in the block, in the KA10 by executing the last word as an instruction.

### 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**

| 70004 | *I* | *X* | *Y* |
|---|---|---|---|
| 0 12 | 13 | 14 17 | 18 35 |

Read the contents of the console data switches into location *E*.

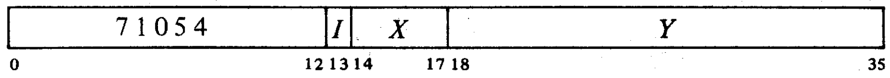
**DATAO PI,** **Data Out, Console**

| 70054 | *I* | *X* | *Y* |
|---|---|---|---|
| 0 12 | 13 | 14 17 | 18 35 |

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).

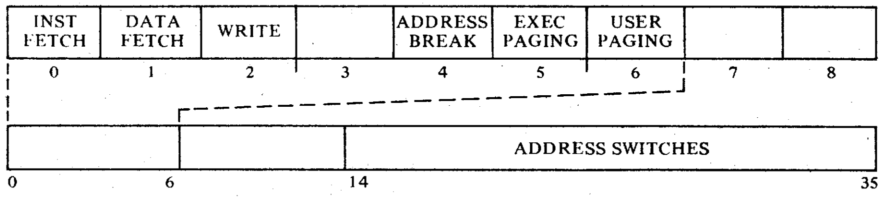
Once the indicators have been loaded by the program, no address condition selected from the console [§§2.18, 2.19] can load them until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key.

**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 §2.19.

## 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, but they 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.

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 part of the conditions given by a CONO (usually bits 33–35). 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).

**Interrupt Requests.** When a device requires service it sends an interrupt request signal over the in-out bus to its assigned channel in the processor. If the channel is on, the processor accepts the request at the next memory access unless the processor is either starting an interrupt on any channel or holding an interrupt on the same channel. The request signal is a level, so it remains on the bus until turned off by the program (CONO, DATAO or

DATAI). Thus if a request is not accepted because of the conditions given above, it will be accepted when those conditions no longer hold. 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.

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.

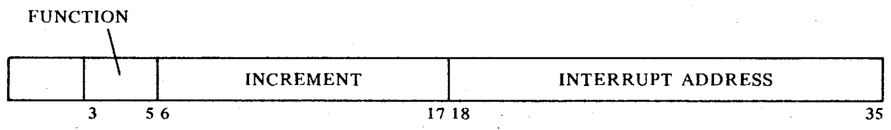
**Starting an Interrupt.** After a request is accepted the channel must wait for the interrupt to start. No interrupts can be started unless the priority interrupt system is active. Furthermore, the processor cannot start an interrupt if it is already holding an interrupt on a channel with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). If there is a higher priority channel waiting, the processor stops the current program to start an interrupt on the waiting channel that has highest priority. The interrupt starts 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), or following a transfer in a BLT. The KI10 can also interrupt the relatively long process of calculating the quotient in double floating division. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

For the KA10 two fixed 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 KA10 starts an interrupt for channel $N$ by executing the instruction in the first interrupt location for the channel, *ie* location $40 + 2N$. Even though the processor may be in user mode when an interrupt occurs, the interrupt operations are performed in executive mode.

Interrupt locations for a second processor are $140 + 2N$ and $141 + 2N$.

The KI10 starts an interrupt by sending an interrupt-granted signal for the channel on which it has accepted a request. This 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. The KI10 also has a pair of fixed locations associated with each channel, and these have the same numbers as in the KA10 but are locations in the executive process table. These 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.

Note that there are therefore two orders of priority associated with a KI10 interrupt: first the channel, and then for all devices requesting interrupts simultaneously on the same channel, proximity to the processor on the bus.



| *Bits 3–5* | *Interrupt Function* |
|---|---|
| 0 | Processor waiting or no response. If the latter, perform a standard interrupt (see function 1). |
| 1 | Standard interrupt – execute the instruction in location $40 + 2N$ 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. This means that for simultaneous requests on a given channel, all KI10 devices have priority over KA10 devices.

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.

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 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 special effects produced by different types of interrupt instructions depend upon the processor.

**KI10 Interrupt Instructions.** Besides instructions, the KI10 can perform other interrupt operations as described above. 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. The interrupt instructions executed in a standard or dispatch interrupt fall into three categories.

♦ *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 nor a BLT will function in a reasonable manner as an interrupt instruction. Therefore do not use them.

**KA10 Interrupt Instructions.** In the KA10 the interrupt instructions fall into two categories.

♦ *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 + 2*N* is a BLKI or BLKO and the count does reach zero, the processor executes the instruction in location 41 + 2*N*. 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 + 2*N* or *any* IO instruction in location 41 + 2*N* hangs up the processor.

**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 active 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 in the KI10, an MUUO. If flag restoration is not desired, a JRST 10, can be used instead.

*Caution*

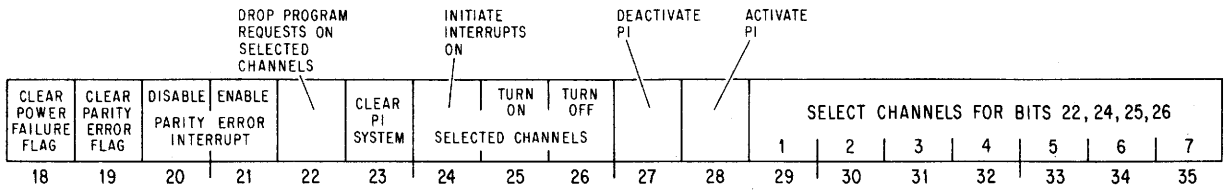
An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its channel and all channels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

**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**

| 70060 | *I* | *X* | *Y* |
|---|---|---|---|
| 0 — 12 | 13 | 14 — 17 | 18 — 35 |

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).



*Notes.*

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.

22 *KI10 only:* 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.

*KA10:* 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.

*KI10:* 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.

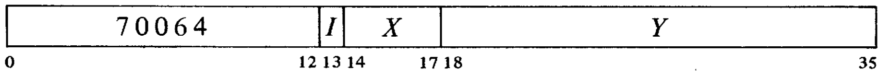
25 Turn on the channels selected by 1s in bits 29–35 so interrupt requests can be accepted on them.

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 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 (as well as several bits of KA10 processor conditions and nine KI10 console operating switches) into location *E* as shown.

| INST FETCH | DATA FETCH | WRITE | ADDRESS STOP | ADDRESS BREAK | EXEC PAGING | USER PAGING | PAR STOP | NXM STOP | | | PROGRAM REQUESTS ON CHANNELS | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

| POWER FAILURE | PARITY ERROR | PARITY ERROR INTERRUPT ENABLED | INTERRUPT IN PROGRESS ON CHANNELS | | | | | | | PI ACTIVE | CHANNELS ON | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |

*Notes.*

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; 1s in bits 11–17 (which are available only in the KI10) 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 priority interrupt system is active.

The remaining conditions read by this instruction have nothing to do with the interrupt. Bits 0–8 are available only in the KI10, where they reflect the settings of various console operating switches; for information on these switches refer to §2.19. Bits 18–20 actually read KA10 processor status conditions [§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.

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 never waits longer than 10 μs with the KI10. With the KA10 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.** 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 a break 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 a KI10 MUUO the routine must save locations 424 and 425 of the user process table. In other cases it is not the pair of consecutive locations that are relevant, as the second contains the instruction to handle the UUO. Thus for a KA10 UUO or a KI10 LUUO the routine must save location 40, unrelocated or in the executive process table respectively, 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.

## 2.14 TRAPPING AND PROCESSOR CONDITIONS

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 various conditions, 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.

For situations of this nature, various internal conditions can interrupt the program. Both processors use condition IO instructions to control the appropriate flags and to inspect other conditions of interest to the program. The KI10 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 priority interrupt system. Violation of instruction restrictions by a user or the supervisor is handled by trapping as an MUUO; violation of memory restrictions is handled as a processor condition in the KA10 (as explained here) but is handled in the KI10 by trapping [§2.15].

### Overflow Trapping (KI10 Only)

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. If overflow traps have been enabled by the Monitor, then 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.

| *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 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, 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.)

should the trap instruction itself set a trap flag (not necessarily the same one), a second trap occurs.

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.

## KI10 Processor Conditions

In the KI10, page failures and overflow are handled by trapping, but there are a number of other internal conditions that can signal the program by requesting an interrupt on a channel assigned to the processor. The program can actually assign two channels – 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]. Inspection of other conditions and control over 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.

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. The program must both enable the auto restart feature and respond to the setting of Power Failure in order for the processor to restart itself. If the program fails to clear Power Failure or enable the auto restart within 4 ms after failure is detected, there is no restart. But if the auto restart is enabled and Power Failure is clear, then when power levels are again adequate the processor will restart itself by executing the instruction in location 70 in kernel mode (provided the power switch is on).

The processor device code is 000, mnemonic APR.

**CONO APR, Conditions Out, Arithmetic Processor**

| | | | |
|---|---|---|---|
| 70020 | *I* | *X* | *Y* |
| 0 12 | 13 | 14 17 | 18 35 |

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).

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | CLEAR NONEXISTENT MEMORY | | | | | | |
| RESET TIMER | CLEAR ALL IN-OUT DEVICES | DISABLE TIMER | ENABLE TIMER | DISABLE AUTO RESTART | ENABLE AUTO RESTART | DISABLE CLOCK INTERRUPT | ENABLE CLOCK INTERRUPT | CLEAR CLOCK | | CLEAR IN-OUT PAGE FAILURE | / | PRIORITY INTERRUPT ASSIGNMENT-ERROR | | | PRIORITY INTERRUPT ASSIGNMENT-CLOCK | | |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |

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**

| | | | |
|---|---|---|---|
| 70024 | *I* | *X* | *Y* |
| 0 12 | 13 | 14 17 | 18 35 |

Read the status of the processor (as well as various console switches and maintenance functions) into location *E* as shown.

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | MAINTENANCE MODE | | | | | | | | | |
| | MEM OVERLAP DISABLE | FM MANUAL | MI PROG DISABLE | CONSOLE DATA LOCK | CONSOLE LOCK | 50 HERTZ | MANUAL MARGINS | / | POWER LOW | MARGIN LOW | | SENSE SWITCHES 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | * | PARITY ERROR INTERRUPT ENABLED | | * | | | CLOCK INTERRUPT ENABLED | * | | * | * NONEXISTENT MEMORY | | | | | | |
| TIME OUT | PARITY ERROR | / | TIMER ENABLED | POWER FAILURE | AUTO RESTART ENABLED | | / | CLOCK | | IN-OUT PAGE FAILURE | / | PRIORITY INTERRUPT ASSIGNMENT-ERROR | | | PRIORITY INTERRUPT ASSIGNMENT-CLOCK | | |
| 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |

*Notes.*

*These bits cause interrupts.

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 §2.19. Bits 7–10 are maintenance functions for which the reader should refer to Chapter 10 of the maintenance manual.

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).

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.

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.

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, at which time PC points to the instruction being performed or to the one following it.

22 Ac power has failed. The program should save PC, the flags, mode information and fast memory in core, and halt the processor.

The setting of this flag requests an interrupt on the error channel. After 4 ms the processor is cleared. But at that time, if Auto Restart Enabled is set and the program has cleared Power Failure (CONO PI,400000), then when adequate power levels are restored, the processor will go back into normal operation in kernel mode at location 70 (provided the power switch is on).

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.

28 A page failure has occurred in an interrupt instruction. The setting of this flag requests an interrupt on the error channel.

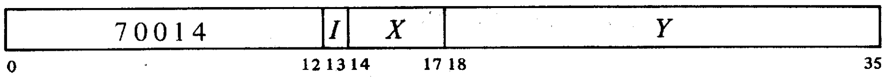
Note: A page failure in an interrupt instruction is regarded as a fatal error, and it causes an interrupt instead of a page failure trap. The kernel mode program is expected to set up the interrupt instructions so that a page failure simply cannot occur.

PC bears no relation to the unanswered reference if the attempted access originated from a console key function.

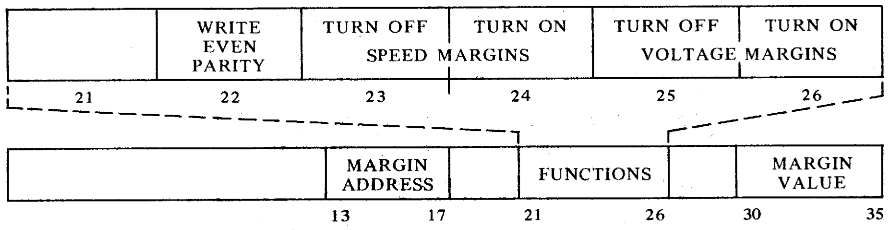
29 The processor attempted to access a memory that did not respond within 100 $\mu$s. The setting of this flag requests an interrupt on the error channel, at which time PC points either to the instruction containing the unanswered reference or to the one following it.

This instruction is for maintenance only. For further information refer to Chapter 10 of the *KI10 Maintenance Manual.*

**DATAO APR, Maintenance Data Out, Arithmetic Processor**



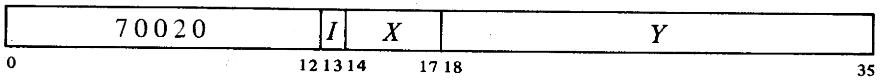
Supply diagnostic information and perform diagnostic functions according to the contents of location *E* as shown.



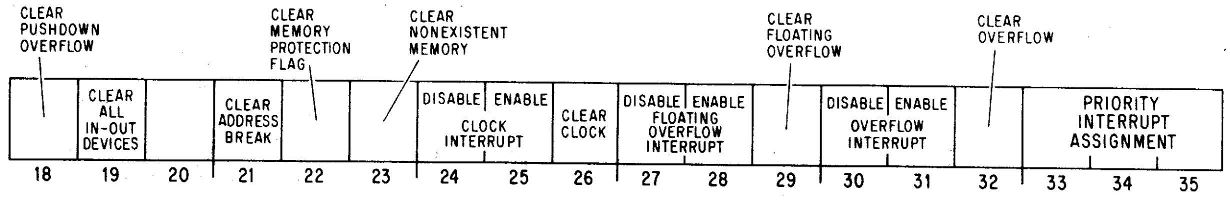
## KA10 Processor Conditions

There are a number of internal conditions that can signal the program by requesting an interrupt on a channel assigned to the processor. 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.

**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).

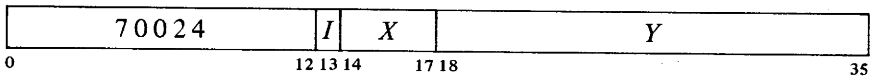


*Notes.*

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.

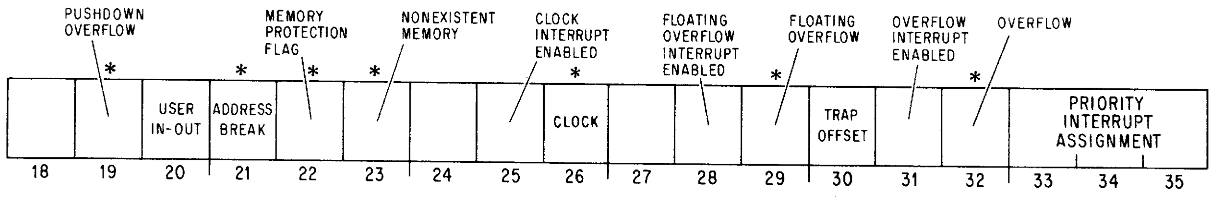
A 1 in bit 19 produces the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system, nor the processor flags cleared by this instruction or CONO PI,).

**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 request interrupts.



*Notes.*

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.

20 User In-out – even if the processor is in user mode, there are no instruction restrictions (but memory restrictions still apply) [§2.16].

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:

PC bears no relation to the break if the access was requested for a console key function.

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.

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.

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.

## 2.15 KI10 MODES

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.

♦ Unless User In-out is set do not give any IO instruction with device code less than 740, 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.

♦ 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

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*].

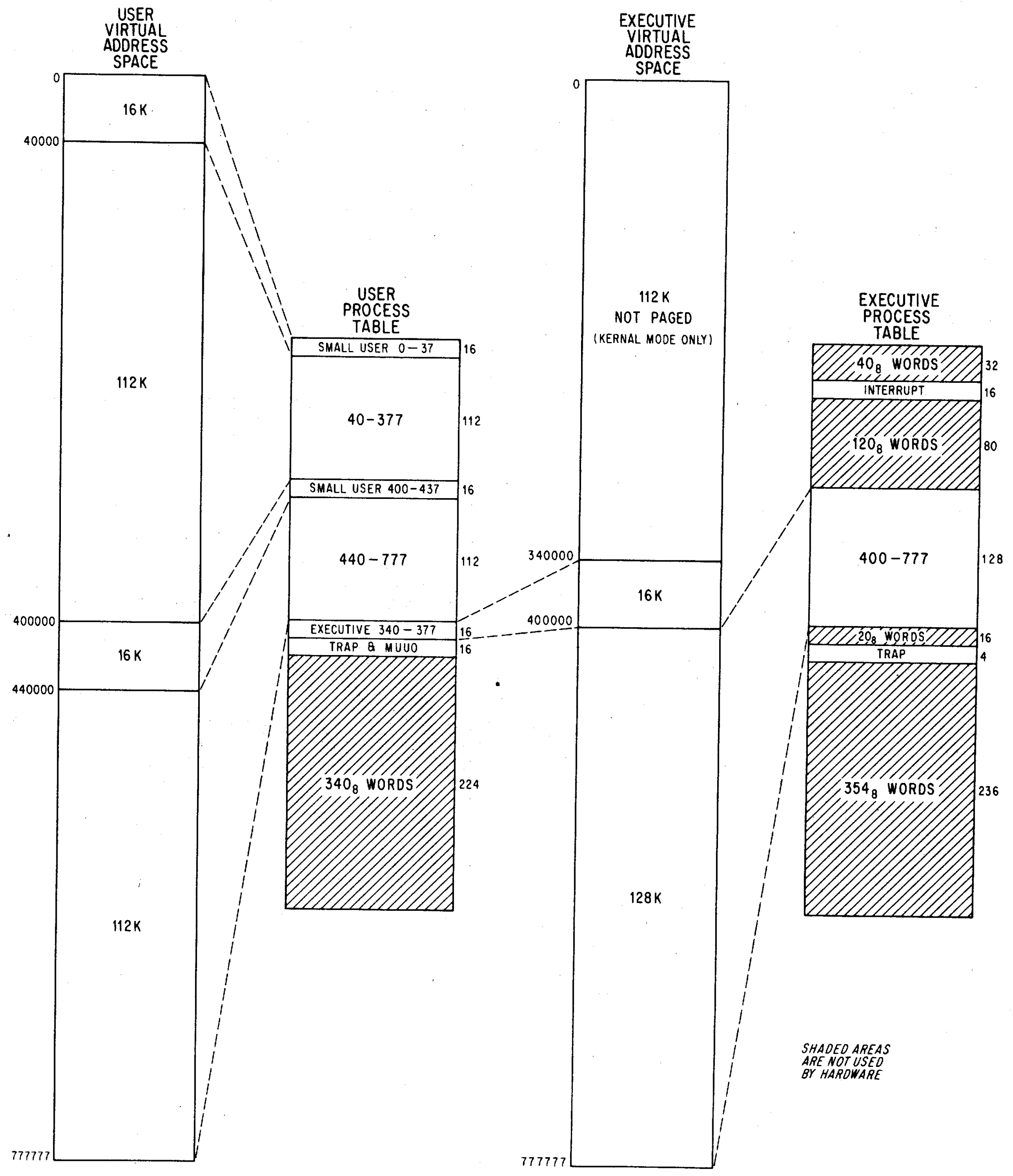
All of memory both virtual and physical is divided into pages of 512 words each. The virtual memory space addressable by a program is 512 pages; the locations in virtual memory are specified by 18-bit addresses, where the left nine bits specify the page number and the right nine the location within the 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.

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.

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.

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.

| DATA FOR EVEN VIRTUAL PAGE | | | | | | DATA FOR ODD VIRTUAL PAGE | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *A* | *P* | *W* | *S* | *X* | PHYSICAL PAGE ADDRESS BITS 14–26 | *A* | *P* | *W* | *S* | *X* | PHYSICAL PAGE ADDRESS BITS 14–26 |
| 0 | 1 | 2 | 3 | 4 | 5 17 | 18 | 19 | 20 | 21 | 22 | 23 35 |

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.

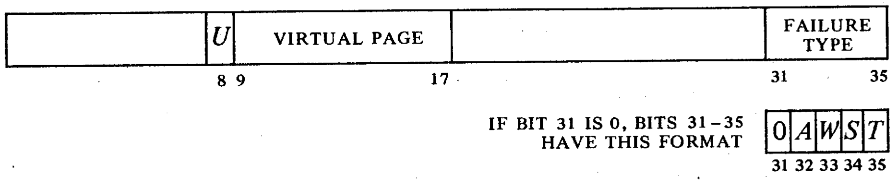
The program can inspect the contents of the page table by using the MAP instruction and IO instructions that address the paging hardware [*see below*].

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.

### 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 (⩾ 20) indicates the type of "hard" failure as follows.

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.

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 requested access to the memory location that was specified by the paging and address switches and for which a comparison was enabled, and the memory reference was for the purpose selected by the address condition switches as follows:

Whether or not a comparison can be made is a function of the settings of the paging switches [§2.19] and the state of the User Address Compare Enable flag [*see below*].

The instruction fetch 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) 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.

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.

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.

Tests for hard page failures are actually made in the order given here.

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 any 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.

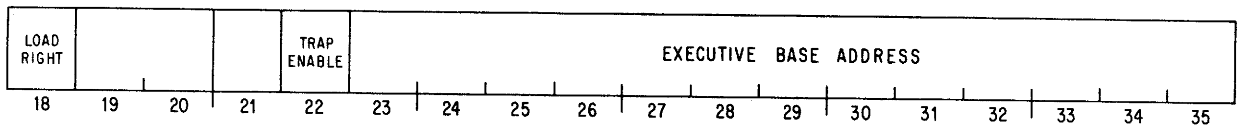
The paging hardware has one non-IO instruction and two condition IO instructions primarily for diagnostic purposes. Otherwise control over the system is exercised by data IO instructions. The device code for the paging hardware is 010, mnemonic PAG.

**DATAO PAG,** **Data Out, Paging**

| 0–12 | 13 | 14–17 | 18–35 |
|---|---|---|---|
| 70114 | *I* | *X* | *Y* |

Set up the paging hardware according to the contents of location *E* as shown.

| 0 | 1–2 | 3 | 4 | 5–17 |
|---|---|---|---|---|
| LOAD LEFT | USER FAST MEMORY BLOCK | SMALL USER | USER ADDRESS COMPARE ENABLE | USER BASE ADDRESS |



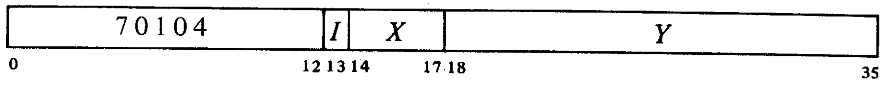
Bits 0 and 18 are change bits. If bit 0 is 0, ignore the rest of the left half word. But if bit 0 is 1, load bits 5–17 into the user base register to select the user process table, select the fast memory block specified by bits 1 and 2 for the user, limit the address space to that of a small user if 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 §2.19.

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 overflow traps if bit 22 is 1.

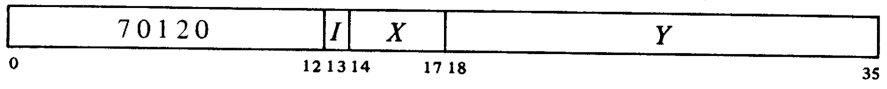
If either bit 0 or 18 is 1, invalidate all data in the associative memory. In other words set the Word Empty bit in each location to indicate that the rest of the word is meaningless and should not be used.

**DATAI PAG,** **Data In, Paging**

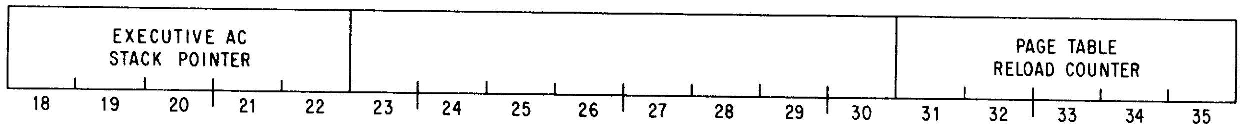


Read the status of the paging hardware into location *E*. The information read is the same as that supplied by a DATAO (bits 0 and 18 are 0).

**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**

| 70124 | *I* | *X* | *Y* |
|---|---|---|---|
| 0–12 | 13 | 14–17 | 18–35 |

This instruction also reads the processor serial number into bits 0–9 of location *E*.

Read the page table reload counter and the contents of the location in the virtual page table specified by it into the right half of location *E* as shown.

| COMPLEMENT OF VIRTUAL PAGE NUMBER | EXECUTIVE ADDRESS SPACE | | WORD EMPTY | PAGE TABLE RELOAD COUNTER |
|---|---|---|---|---|
| 18–26 | 27 | 28–29 | 30 | 31–35 |

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.

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.

**MAP Map an Address**

| 257 | *A* | *I* | *X* | *Y* |
|---|---|---|---|---|
| 0–8 | 9–12 | 13 | 14–17 | 18–35 |

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.

| PAGE FAILURE | *P* | *W* | *S* | NO MATCH | PHYSICAL PAGE ADDRESS BITS 14–26 |
|---|---|---|---|---|---|
| 18 | 19 | 20 | 21 | 22 | 23–35 |

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.

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*.

Read the next four paragraphs very carefully (reading them two or three times is highly recommended).

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,342000]
```

and restores them with these two.

```
MOVSI   T,342000      ;From 342000 to 0
XCT     2,[BLT  T,17]
```

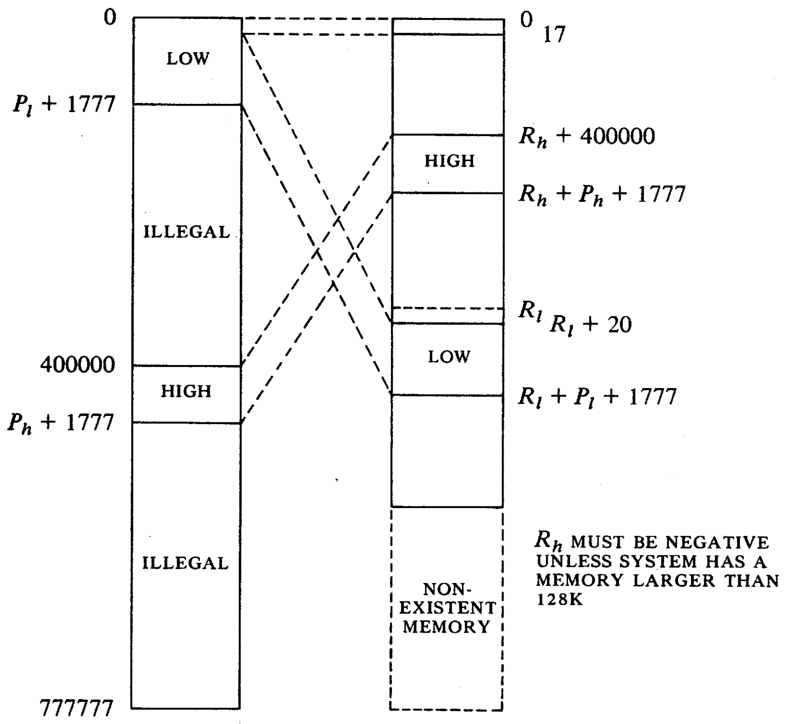
## 2.16 KA10 MODES

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

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.



USER ADDRESSES BEFORE RELOCATION TYPICAL PHYSICAL ADDRESS CONFIGURATION AFTER RELOCATION

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 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 ($P_l > 400000$), 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].

**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.*]

◆ 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.

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.

◆ 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.

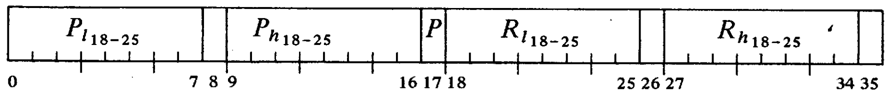
**DATAO APR, Data Out, Arithmetic Processor**

| 70014 | $I$ | $X$ | $Y$ |
|---|---|---|---|
| 0 … 12 | 13 | 14 … 17 | 18 … 35 |

Load the protection and relocation registers from the contents of location

For a two part nonreentrant program, set $P = 0$. For a one-part nonreentrant program, make $P_h \leqslant P_l$. 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.

$E$ 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.

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.

## 2.17 REAL TIME CLOCK DK10

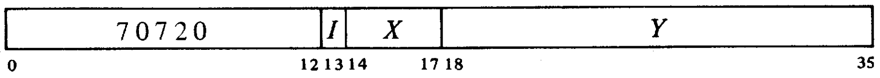
The clock referred to throughout this section is the DK10 real time clock and should not be confused with the line frequency clock whose flag is one of the processor conditions [§2.14].

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 $\mu$s, the total count is about 2.6 seconds.

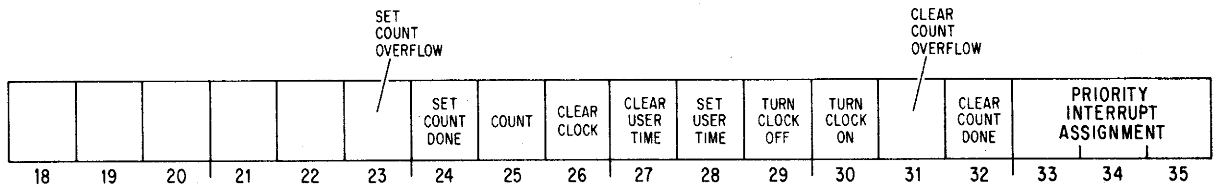
The program turns the clock on and off by enabling and disabling the counter. The clock has two modes of operation: with the User Time flag clear, the counter operates continuously; with User Time set, the counter stops while the processor is handling interrupts. Hence in the latter mode the clock discounts 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**



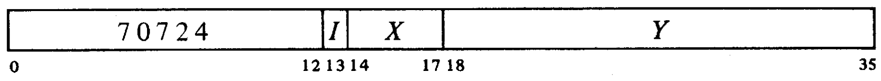
Assign the interrupt channel specified by bits 33–35 of the effective conditions $E$ and perform the functions specified by bits 23–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



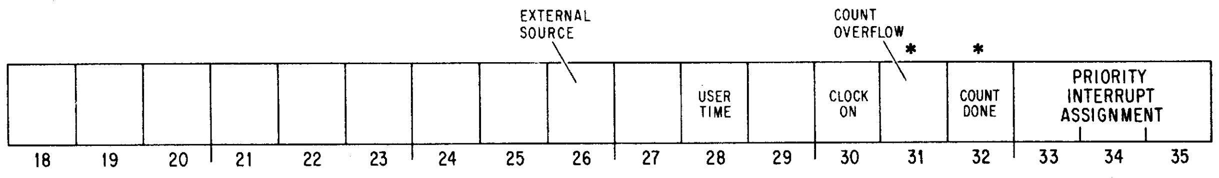
A 1 in bit 26 clears the clock counter and the Count Done, Count Overflow and User Time flags, turns off the clock, and dismisses the PI assignment (assigns zero). The effect of giving conflicting conditions is indeterminate.

**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.



*Notes.*

***These bits request interrupts.**

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).

Note that to time a user properly, the Monitor must also compensate for any noninterrupt time taken from the user.

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.

**DATAO CLK,** **Data Out, Clock**

| 70714 | *I* | *X* | *Y* |
|---|---|---|---|
| 0 12 | 13 | 14 17 | 18 35 |

The comparison of the counter against the interval register that follows every count is inhibited while this instruction is loading the register.

Load the contents of the right half of location *E* into the interval register.

**DATAI CLK,** **Data In, Clock**

| 70704 | *I* | *X* | *Y* |
|---|---|---|---|
| 0 12 | 13 | 14 17 | 18 35 |

The counter is always stable while being read, and any count held back is picked up immediately afterward.

Read the current contents of the clock counter into the right half of location *E*.

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.

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.

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

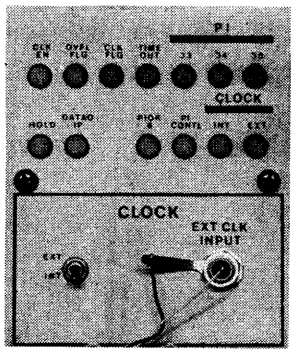
$$T(C + nI)$$

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 $n \leqslant 2^{18}$ 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.**

**Operation.** The KI10 clock, which is usually installed in a DECtape cabinet, has a small control panel mounted directly on the logic behind the cabinet doors. In the lower part of the panel is a switch for selecting the internal source or an external input from the BNC connector at the right. The external input must be supplied through a 100 ohm coaxial cable and must have a frequency no greater than 400 kHz; its triggering voltage change must be from −3 volts to ground. If the input is a pulse train, the minimum pulse width is 100 ns. If the input is a sequence of level changes, it must have a minimum low level (−3 volts) duration of 400 ns before each positive-going change, a rise time of 60 ns maximum, and a high level duration of 40 ns minimum.

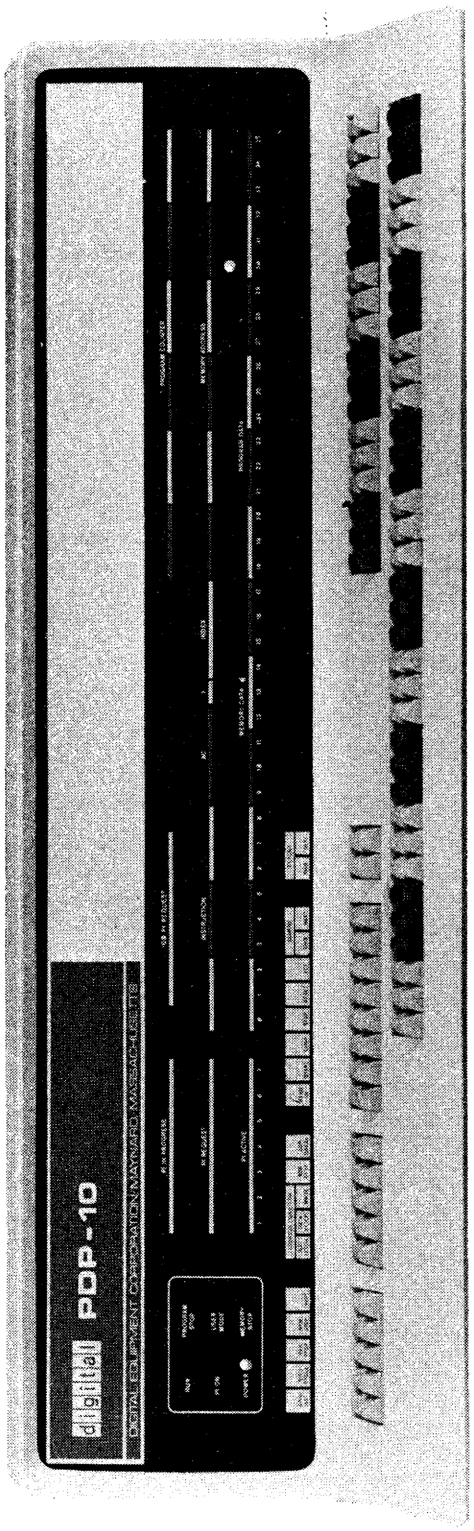


Clock Control Panel

The leftmost light in the upper row at the top of the panel indicates when the clock is on (*ie* when the counter is enabled). The next two lights are the Count Overflow and Count Done flags. TIME OUT indicates when the numbers in the interval register and the clock counter are identical – this light goes out as soon as either changes state. The remaining lights in the upper row are the PI assignment. The two lights at the left in the lower row display signals that synchronize the DATAI and DATAO to the clock so that counting is postponed while the counter is being read and there is no sampling while the interval is being loaded. PIOK8 is a processor-generated signal which indicates that there is no interrupt being held and no channel waiting for an interrupt; the next light is the User Time flag. The final two lights indicate the origin of the clock source.

## 2.18 KA10 OPERATION

Most of the controls and indicators used for normal operation of the processor and for program debugging are located on the console operator panel shown here. The indicators are on the vertical part of the panel; in front of them are two rows of two-position keys and switches (keys are momentary contact, switches are alternate action). A key or switch is on or represents a 1 when the front part is down.



The thirty-six switches in the front row and the eighteen at the right in the back row are respectively the data and address switches through which the operator can supply words and addresses for the program and for use in conjunction with the operating keys and switches. The correspondence of switches to bit positions is indicated by the numbers at the bottom row of lights. At the left end of the back row are ten operating switches, which supply continuous control levels to the processor. At their right are ten operating keys, which initiate or terminate operations in the processor. The names of the operating keys and switches appear on the vertical part of the panel below the lights.

Also of interest to the operator is the small panel shown opposite, which is located above the main panel at the left of the tape reader. The upper section of this panel contains a total hours meter and the margin-check controls. The lower section contains the power switch, speed controls for slowing down the program, switches to select the device for readin mode (lower part in represents a 1), and four additional operating switches. The normal position for these last four is with the upper part in; in this position FM ENB (fast memory enable) is on, the others are all off.

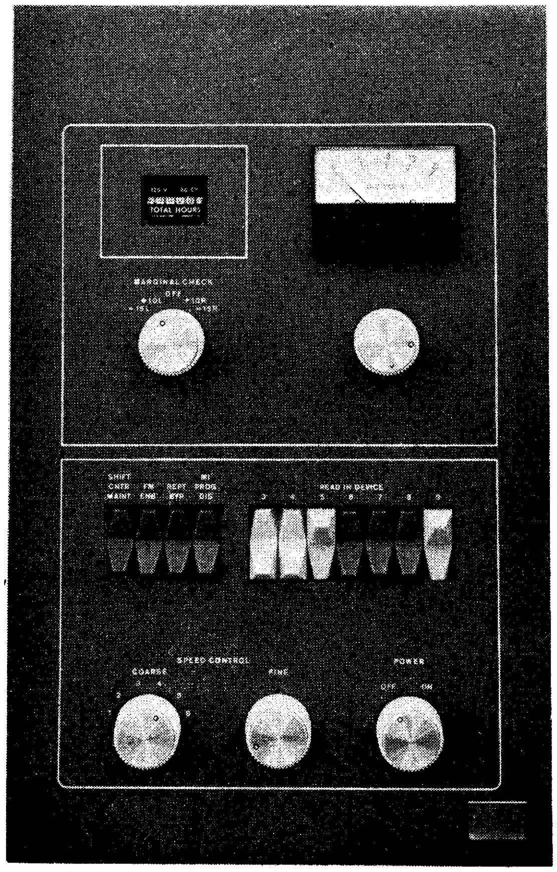
### Indicators

When any indicator is lit the associated flipflop is 1 or the associated function is true. Some indicators display useful information while the processor is running, but many change too frequently and can be discussed only in terms of the information they display when the processor is stopped. The program can stop the processor only at the completion of the HALT instruction; the operator can stop it at the end of every instruction or memory reference, or for maintenance purposes, after every step in any operation that uses the shift counter (shifting, multiplication, division, byte manipulation).

Of the long rows of lights at the right on the operator panel, the top row displays the contents of PC, the middle row displays the instruction being executed or just completed, and the bottom row are the memory indicators. The right half of the middle row displays MA, the left half displays IR [*see page 1-2*]. In an IO instruction the left three instruction lights are on, the remaining instruction lights and the left AC light are the device code, and the remaining AC lights complete the instruction code. The I, index and MA lights change with each indirect reference in an effective address calculation; at the end of an instruction I is always off.

Above the memory indicators appear two pairs of words, PROGRAM DATA and MEMORY DATA. If the triangular light beside the former pair is on, the indicators display a word supplied by a DATAO PI,; if any other data is displayed the light beside MEMORY DATA is on instead. While the processor is running the physical addresses used for memory reference (the relocated address whenever relpcation is in effect) are compared with the contents of the address switches. Whenever the two are equal the contents of the addressed location are displayed in the memory indicators. However, once the program loads the indicators, they can be changed only by the program until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key (see below).

The four sets of seven lights at the left display the state of the priority interrupt channels [*see pages 2-81 to 2-85*]. The PI ACTIVE lights indicate which channels are on. The IOB PI REQUEST lights indicate which channels are receiving request signals over the in-out bus; the PI REQUEST lights indicate channels on which the processor has accepted requests. Except in the case of a program-initiated interrupt, a REQUEST light can go on only if the corresponding ACTIVE light is on. The PI IN PROGRESS lights indicate channels on which interrupts are currently being held; the channel that is actually being serviced is the lowest-numbered one whose light is on. When a PROGRESS light goes on, the corresponding REQUEST goes off and cannot go on again until PROGRESS goes off when the interrupt is dismissed.



*Above:* Margin Check and Maintenance Panel
*Opposite:* Console Operator Panel

At the left end of the panel are a power light and these control indicators.

NOTE: If a REQUEST light stays on indefinitely with the associated PROGRESS light off and PC is static, check the PI CYC light on the indicator panel at the top of bay 2. If it is on, a faulty program has hung up the processor. Press STOP.

RUN
The processor is in normal operation with one instruction following another. When the light goes off, the processor stops.

PI ON
The priority interrupt system is active so interrupts can be started (this corresponds to CONI PI, bit 28).

PROGRAM STOP
IR now contains a HALT instruction. If RUN is off, MA displays 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 presses the continue key).

If RUN and PROGRAM STOP are both on, the processor is probably in an indirect address loop. Press STOP.

USER MODE
The processor is in user mode (this corresponds to bit 5 of a PC word).

MEMORY STOP
The processor has stopped at a memory reference. This can be due to single cycle operation, satisfaction of an address condition selected at the console, reference to a nonexistent memory location, or detection of a parity error.

The remaining processor lights are on the indicator panels at the tops of the bays [*illustrated on page F2*]. Bay 2 displays AR, BR and MQ, the output of the AR adder, and the parity buffer which receives every word transmitted over the memory bus. The RL and PR lights at the lower right display the relocation and protection registers for the low part of the area assigned to a user program and the left eight bits of the relocated address for that part. The remaining lights are for maintenance.

The upper four rows on the bay 1 panel include the indicators for reader, punch and teletype, which are described in Chapter 3. The bottom row displays the information on the data lines in the IO bus. The AR lights at the upper right are the flags – FXU is Floating (exponent) Underflow, DCK is No Divide (divide check). OV COND is the condition that the 0 and 1 carries are different, *ie* the condition that indicates overflow. The First Part Done flag is BYF6 in the MISC lights in the top row; User In-out is IOT USER in the EX lights at the center of the panel. The CPA lights in the top row and the five lights under them at the left are the processor conditions – PDL OV is Pushdown (list) Overflow. The AS= lights in the middle row indicate when the (relocated) core memory address or the fast memory address is the same as the address switches. The remaining lights are for maintenance.

The panels on the memories are shown in Appendix F. These are almost exclusively for maintenance, and (as with most of the lights on the processor bays) if the operator must use them he should consult the maintenance manual and the flow charts. The ACTIVE lights indicate which processor currently has access to the memory.

## Operating Keys

*Caution*
*Never* press two keys simultaneously as the processor may attempt to perform both functions at once.

Each key except STOP turns on one of the key indicators at the upper right on the bay 2 panel. These are for flipflops that allow the key functions to be repeated automatically and also allow certain of them to be synchronized to the processor time chain so they can be performed while the processor is running.

If RUN is on, pressing this key has no effect.

READ IN
Clear all IO devices and all processor flags including User; turn on the RIM light in the upper right on bay 1 and the KEY RDI light in the upper right

on bay 2. Execute DATAI *D*,0 where *D* is the device code specified by the readin device switches on the small panel at the left of the reader. Then execute a series of BLKI *D*,0 instructions until the left half of location 0 reaches zero, at which time turn off RIM and KEY RDI. Stop only if the single instruction switch is on; otherwise turn on RUN and execute the last word read as an instruction. [*For information on the data format refer to page 2-79.*]

The rightmost device switch is for bit 9 of the instruction and thus selects the least significant octal digit (which is always 0 or 4) in the device code.

*CAUTION*

Do not initiate any other key function while RIM is on. If read in must be stopped (*eg* because of a crumpled tape), press RESET (see below).

START
Load the contents of the address switches into PC, turn on RUN, and begin normal operation by executing the instruction at the location specified by PC.

This key function does not disturb the flags or the IO equipment; hence if USER MODE is lit a user program can be started.

If RUN is on, pressing this key has no effect.

CONT (Continue)
Turn on RUN (if it is off) and begin normal operation in the state indicated by the lights.

STOP
Turn off RUN so the processor stops before beginning the next instruction. Thus the processor usually stops at the end of the current instruction, which is displayed in the lights. However, if a key function that can be performed while RUN is on has been synchronized, the processor performs that function before stopping. In either case PC points to the next instruction.

If the processor does not reach the end of the instruction within 100 $\mu$s, inhibit further effective address calculation – it is assumed the processor is caught in an indirect addressing loop. Pressing CONT when the processor is stopped in an address loop causes it to start the same instruction over.

RESET
Clear all IO devices and clear the processor including all flags. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on duplicate the action of the STOP key before clearing.

If STOP will not stop the processor, pressing this key will.

XCT
Execute the contents of the data switches as an instruction without incrementing PC. If RUN is on, insert this instruction between two instructions in the program. Inhibit priority interrupts during its execution to guarantee that it will be finished.

If USER MODE is lit all user restrictions apply to an instruction executed from the console.

Note that an instruction executed from the console can alter the processor state just like any instruction in the program: it can change PC by jumping or skipping, alter the flags, or even cause a nonexistent-memory stop.

NOTE

The remaining key functions all reference memory. They use an absolute address and all of memory is available to them; in other words protection and relocation are not in effect even if USER MODE is lit. However they can set such flags as Address Break and Nonexistent Memory.

EXAMINE THIS
Display the contents of the address switches in the MA lights and the contents of the location specified by the address switches in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on, insert this function between two instructions in the program.

If RUN is on, pressing this key has no effect.

EXAMINE NEXT
Add 1 to the address displayed in the MA lights and display the contents of the location specified by the incremented address in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA).

DEPOSIT
Deposit the contents of the data switches in the location specified by the address switches. Display the address in the MA lights and the word deposited in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on, insert this function between two instructions in the program.

If RUN is on, pressing this key has no effect.

DEPOSIT NEXT
Add 1 to the address displayed in the MA lights and deposit the contents of the data switches in the location specified by the incremented address. Display the word deposited in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA).

### Operating Switches

Whenever the processor references memory at the location specified by the address switches (relocated if USER MODE is on), the contents of that location are displayed in the memory indicators (unless the light beside PROGRAM DATA is on). The group of five switches at the left of the keys allows the operator to make the processor halt or request an interrupt when reference is made to the specified location *in core memory* for a particular purpose (no action is produced by fast memory reference). The purpose is selected by the three address condition switches. INST FETCH selects the condition that access is 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. DATA FETCH selects access for retrieval of an operand (other than in an XCT). WRITE selects access for writing in memory. Whenever reference to the specified location satisfies any selected address condition, the processor performs the action selected by the other two switches. ADR STOP halts the processor with MEMORY STOP on (PC points to the instruction that was being executed, or if the MC WR light on bay 2 is on, PC may point to the one following it); ADR BREAK turns on the CPA ADR BRK light (Address Break flag, CONI APR, bit 21) on bay 1, requesting an interrupt on the processor channel.

AC and index register references can be included by turning off the FM ENB switch (see below).

The description of each switch relates the action it produces while it is on.

SING INST
Whenever the processor is placed in operation, clear RUN so that it stops at the end of the first instruction. Hence the operator can step through a program one instruction at a time, by pressing START for the first one and CONT for subsequent ones. Each time the processor stops, the lights display the same information as when STOP is pressed.

CLK FLAG (Clock flag) on bay 1 is held off to prevent clock interrupts while SING INST is on. Otherwise interrupts would occur at a faster rate than the instructions.

SING INST will not stop the processor if a hangup prevents it from getting to the end of an instruction. Use STOP or RESET.

SING CYCLE
Whenever the processor is placed in operation, stop it with MEMORY STOP on at the end of the first *core memory* reference. Hence the operator can step through a program one memory reference at a time, by pressing START for the first one and CONT for subsequent ones. To determine what information is displayed in the lights, consult the flow charts.

To stop at AC and index register references, turn off the FM ENB switch (see below).

PAR STOP
Stop with MEMORY STOP on at the end of any memory reference in which even parity is detected in a word read. A parity stop is indicated by the following: CPA PAR ERR (Parity Error flag) on bay 1 is on; and among the PAR lights in the bottom row on bay 2, IGN (ignore parity) and ODD are off, STOP is on, and BIT displays the parity bit for the word in the parity buffer at the left.

If IGN is on (it displays a signal from the memory), parity errors are not detected and no stop can occur.

NXM STOP
Stop with MEMORY STOP on if a memory reference is attempted but the memory does not respond within 100 μs. This type of stop is indicated by CPA NXM FLAG (Nonexistent Memory flag) on bay 1 being on.

REPT
If any key (except STOP) is pressed, then every time the key function is finished, wait a period of time determined by the setting of the speed control and repeat the given key function. If CONT is pressed and no switch is on that would stop the program (*eg* SING INST, SING CYCLE), then continue following the repeat delay whenever a HALT instruction is executed. Continue to repeat the key function until RESET is pressed or REPT is turned off.

The key function is repeated once after REPT is turned off, but this is noticeable only with very long repeat delays.

The end of a key function is equivalent to completion of all processor operations associated with the function only for read in, examine, examine next, deposit, and deposit next. In other cases the processor continues in operation. *Eg* the execute function is finished once the instruction to be executed is set up internally, but the processor then executes that instruction. Hence when using speed range 6, the operator must be careful not to allow the key function to restart before the processor is really finished.

The speed control includes a six-position switch that selects the delay range and a potentiometer for fine adjustment within the range. Delay ranges are as follows.

| *Position* | *Range* |
|---|---|
| 1 | 270 ms to 5.4 seconds |
| 2 | 38 ms to 780 ms |
| 3 | 3.9 ms to 78 ms |
| 4 | 390 μs to 7.8 ms |
| 5 | 27 μs to 540 μs |
| 6 | 2.2 μs to 44 μs |

MI PROG DIS
Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA) and inhibit the program from displaying any information in the memory indicators. The indicators will thus continually display the contents of locations selected from the console.

REPT BYP
If REPT is on, trigger the repeat delay at the *beginning* of the key function. Hence the function is repeated even if it does not run to completion.

FM ENB
This switch is left on for normal operation with a fast memory. Turning it *off* (lower part in) substitutes the first sixteen core locations for the fast memory. The switch is left off if there is no fast memory, and it can be used to allow stopping or breaking at fast memory references.

SHIFT CNTR MAINT
Stop before each step in any shift operation. Pressing CONT resumes the operation. Once a shift has been stopped, the processor will continue to stop at each step throughout the rest of the given shift operation even if the switch is turned off.

At the right end of panel 1J behind the bay doors are two toggle switches. FP TRP causes the floating point and byte manipulation instructions (codes 130–177) to trap to locations 60-61. MA TRP OFFSET moves the trap and interrupt locations to 140–161 for a second processor connected to the same memory.

Inside each memory bay are switches for selecting the memory number and interleaving memories. Also in the memory are a power switch, a restart pushbutton, and a switch for single step operation (these three are located on the indicator panel for the MB10 memory).

**NOTE**
**Information on the KI10 operation is not available at this time.**

# Appendices

# APPENDIX A

## INSTRUCTION AND DEVICE MNEMONICS

The illustration on the next page shows the derivation of the instruction mnemonics. The two tables following it 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 table includes 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.

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 option 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 A11 is a list of all instructions showing their actions in symbolic form.

```
     ⎧ E          ⎫
MOV  ⎨ e Negative ⎬──────────────────┐  ⎧ to AC
     ⎪ e Magnitude⎪                  │  ⎪ Immediate to AC
     ⎩ e Swapped  ⎭                  ├──⎨ to Memory
                          ⎧ no effect  ⎫ │  ⎩ to Self
Half word ⎧Right⎫ to ⎧Right⎫ ⎪ Ones       ⎬─┘
          ⎩Left ⎭    ⎩Left ⎭ ⎪ Zeros      ⎪
                          ⎩ Extend sign⎭
BLock Transfer
EXCHange AC and memory
```

```
use present pointer⎫ and ⎧ LoaD Byte into AC
Increment pointer  ⎭     ⎩ DePosit Byte in memory
Increment Byte Pointer
```

```
PUSH down⎫ ⎧ ~
POP up   ⎭ ⎩ and Jump
```

```
        ⎧ Zeros                 ⎫
        ⎪ Ones                  ⎪
SET to  ⎨ Ac                    ⎬──────────┐
        ⎪ Memory                ⎪          │
        ⎪ Complement of Ac      ⎪          │
        ⎩ Complement of Memory  ⎭          │   ⎧ AC
                                           │   ⎪ AC Immediate
             ⎧ ~                         ⎫ ├to ⎨ Memory
AND          ⎪ with Complement of Ac     ⎬─┤   ⎩ Both
inclusive OR ⎨ with Complement of Memory ⎪ │
             ⎩ Complements of Both       ⎭ │
Inclusive OR ⎫                             │
eXclusive OR ⎬─────────────────────────────┘
EQuiValence  ⎭
```

```
SKIP if memory⎫                             ⎧ never
JUMP if AC    ⎭─────────────────────────┐   ⎪ Less
                                        │   ⎪ Equal
Add One to       ⎫ ⎧ memory and Skip⎫ if┤   ⎪ Less or Equal
Subtract One from⎭ ⎩ AC and Jump    ⎭   │   ⎨ Always
                                        │   ⎪ Greater
Compare Ac ⎧Immediate  ⎫ and skip if AC─┘   ⎪ Greater or Equal
           ⎩with Memory⎭                    ⎩ Not equal

Add One to Both halves of AC and Jump if ⎧ Positive
                                         ⎩ Negative
```

```
Arithmetic SHift⎫ ⎧ ~
Logical SHift   ⎬ ⎩ Combined
ROTate          ⎭
```

```
ADD               ⎫
SUBtract          ⎪
MULtiply          ⎪
Integer MULtiply  ⎬───────────┐  ⎧ ~
DIVide            ⎪           ├──⎨ Immediate
Integer DIVide    ⎭           │  ⎪ to Memory
                   ┌and Round─┘  ⎩ to Both
Floating AdD      ⎫│
Floating SuBtract ⎬┴───────────  ⎧ ~
Floating MultiPly ⎪              ⎨ Long
Floating DiVide   ⎭              ⎪ 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
```

```
      ⎧ to SubRoutine
      ⎪ and Save Pc
      ⎪ and Save Ac
      ⎪ and Restore Ac
      ⎪ if Find First One
      ⎪ on Flag and CLear it
Jump  ⎨ 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 ⎫──┐  ⎧ In
BLocK⎭  ├──⎩ Out
CONditions┤
          └in and Skip if ⎧ all masked bits Zero
                          ⎩ some masked bit One
```

```
        ⎧ with Direct mask  ⎫ ⎧ No modification         ⎫          ⎧ never
Test AC ⎨ with Swapped mask ⎬ ⎨ set masked bits to Zeros⎬ and skip ⎨ if all masked bits Equal 0
        ⎪ Right with E      ⎪ ⎪ set masked bits to Ones ⎪          ⎪ if Not all masked bits equal 0
        ⎩ Left with E       ⎭ ⎩ Complement masked bits  ⎭          ⎩ Always
```

## INSTRUCTION MNEMONICS

## NUMERIC LISTING

| | | | | | |
|---|---|---|---|---|---|
| 000 | ILLEGAL | 106 | | 162 | FMPM |
| 001 | LUUO'S | 107 | | 163 | FMPB |
| . | | 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 | RESERVED FOR SPECIAL MONITORS | 115 | | 171 | FDVL |
| 043 | | 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 | RESERVED FOR DEC | 125 | ‡DMOVNM | 201 | MOVEI |
| 053 | | 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 |

A4

MNEMONICS

| | | | | | |
|---|---|---|---|---|---|
| 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 |

NUMERIC LISTING

| | | | | | |
|---|---|---|---|---|---|
| 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 |

A6 MNEMONICS

| | | | | | |
|---|---|---|---|---|---|
| 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 |

| | | | | | |
|---|---|---|---|---|---|
| ‡DMOVE | 120 | FSBRB | 157 | HRLS | 507 |
| ‡DMOVEM | 124 | FSBRI | 155 | HRLZ | 514 |
| ‡DMOVN | 121 | FSBRM | 156 | HRLZI | 515 |
| ‡DMOVNM | 125 | FSC | 132 | HRLZM | 516 |
| DPB | 137 | *GETSTS | 062 | HRLZS | 517 |
| †DPC | 250 | HALT | 25420 | HRR | 540 |
| †DSI | 464 | HLL | 500 | HRRE | 570 |
| †DSK | 170 | HLLE | 530 | HRREI | 571 |
| †DSS | 460 | HLLEI | 531 | HRREM | 572 |
| †DTC | 320 | HLLEM | 532 | HRRES | 573 |
| †DTS | 324 | HLLES | 533 | HRRI | 541 |
| *ENTER | 077 | HLLI | 501 | HRRM | 542 |
| EQV | 444 | HLLM | 502 | HRRO | 560 |
| EQVB | 447 | HLLO | 520 | HRROI | 561 |
| EQVI | 445 | HLLOI | 521 | HRROM | 562 |
| EQVM | 446 | HLLOM | 522 | HRROS | 563 |
| EXCH | 250 | HLLOS | 523 | HRRS | 543 |
| FAD | 140 | HLLS | 503 | HRRZ | 550 |
| FADB | 143 | HLLZ | 510 | HRRZI | 551 |
| FADL | 141 | HLLZI | 511 | HRRZM | 552 |
| FADM | 142 | HLLZM | 512 | HRRZS | 553 |
| FADR | 144 | HLLZS | 513 | IBP | 133 |
| FADRB | 147 | HLR | 544 | IDIV | 230 |
| FADRI | 145 | HLRE | 574 | IDIVB | 233 |
| FADRM | 146 | HLREI | 575 | IDIVI | 231 |
| FDV | 170 | HLREM | 576 | IDIVM | 232 |
| FDVB | 173 | HLRES | 577 | IDPB | 136 |
| FDVL | 171 | HLRI | 545 | ILDB | 134 |
| FDVM | 172 | HLRM | 546 | IMUL | 220 |
| FDVR | 174 | HLRO | 564 | IMULB | 223 |
| FDVRB | 177 | HLROI | 565 | IMULI | 221 |
| FDVRI | 175 | HLROM | 566 | IMULM | 222 |
| FDVRM | 176 | HLROS | 567 | *IN | 056 |
| ‡FIX | 122 | HLRS | 547 | *INBUF | 064 |
| ‡FIXR | 126 | HLRZ | 554 | *INIT | 041 |
| ‡FLTR | 127 | HLRZI | 555 | *INPUT | 066 |
| FMP | 160 | HLRZM | 556 | IOR | 434 |
| FMPB | 163 | HLRZS | 557 | IORB | 437 |
| FMPL | 161 | HRL | 504 | IORI | 435 |
| FMPM | 162 | HRLE | 534 | IORM | 436 |
| FMPR | 164 | HRLEI | 535 | JCRY | 25530 |
| FMPRB | 167 | HRLEM | 536 | JCRY0 | 25520 |
| FMPRI | 165 | HRLES | 537 | JCRY1 | 25510 |
| FMPRM | 166 | HRLI | 505 | JEN | 25460 |
| FSB | 150 | HRLM | 506 | JFCL | 255 |
| FSBB | 153 | HRLO | 524 | JFFO | 243 |
| FSBL | 151 | HRLOI | 525 | JFOV | 25504 |
| FSBM | 152 | HRLOM | 526 | JOV | 25540 |
| FSBR | 154 | HRLOS | 527 | JRA | 267 |

A8

MNEMONICS

| | | | | | |
|---|---|---|---|---|---|
| JRST | 254 | ORCAI | 455 | SETOM | 476 |
| JRSTF | 25410 | ORCAM | 456 | *SETSTS | 060 |
| JSA | 266 | ORCB | 470 | SETZ | 400 |
| JSP | 265 | ORCBB | 473 | SETZB | 403 |
| JSR | 264 | ORCBI | 471 | SETZI | 401 |
| JUMP | 320 | ORCBM | 472 | SETZM | 402 |
| JUMPA | 324 | ORCM | 464 | SKIP | 330 |
| JUMPE | 322 | ORCMB | 467 | SKIPA | 334 |
| JUMPG | 327 | ORCMI | 465 | SKIPE | 332 |
| JUMPGE | 325 | ORCMM | 466 | SKIPG | 337 |
| JUMPL | 321 | ORI | 435 | SKIPGE | 335 |
| JUMPLE | 323 | ORM | 436 | SKIPL | 331 |
| JUMPN | 326 | *OUT | 057 | SKIPLE | 333 |
| LDB | 135 | *OUTBUF | 065 | SKIPN | 336 |
| *LOOKUP | 076 | *OUTPUT | 067 | SOJ | 360 |
| †LPT | 124 | †PAG | 010 | SOJA | 364 |
| LSH | 242 | †PI | 004 | SOJE | 362 |
| LSHC | 246 | †PLT | 140 | SOJG | 367 |
| ‡MAP | 257 | POP | 262 | SOJGE | 365 |
| †MDF | 260 | POPJ | 263 | SOJL | 361 |
| MOVE | 200 | PORTAL | 25404 | SOJLE | 363 |
| MOVEI | 201 | †PTP | 100 | SOJN | 366 |
| MOVEM | 202 | †PTR | 104 | SOS | 370 |
| MOVES | 203 | PUSH | 261 | SOSA | 374 |
| MOVM | 214 | PUSHJ | 260 | SOSE | 372 |
| MOVMI | 215 | *RELEAS | 071 | SOSG | 377 |
| MOVMM | 216 | *RENAME | 055 | SOSGE | 375 |
| MOVMS | 217 | ROT | 241 | SOSL | 371 |
| MOVN | 210 | ROTC | 245 | SOSLE | 373 |
| MOVNI | 211 | RSW | 70004 | SOSN | 376 |
| MOVNM | 212 | SETA | 424 | *STATO | 061 |
| MOVNS | 213 | SETAB | 427 | *STATUS | 062 |
| MOVS | 204 | SETAI | 425 | *STATZ | 063 |
| MOVSI | 205 | SETAM | 426 | SUB | 274 |
| MOVSM | 206 | SETCA | 450 | SUBB | 277 |
| MOVSS | 207 | SETCAB | 453 | SUBI | 275 |
| *MTAPE | 072 | SETCAI | 451 | SUBM | 276 |
| †MTC | 220 | SETCAM | 452 | TDC | 650 |
| †MTM | 230 | SETCM | 460 | TDCA | 654 |
| †MTS | 224 | SETCMB | 463 | TDCE | 652 |
| MUL | 224 | SETCMI | 461 | TDCN | 656 |
| MULB | 227 | SETCMM | 462 | TDN | 610 |
| MULI | 225 | SETM | 414 | TDNA | 614 |
| MULM | 226 | SETMB | 417 | TDNE | 612 |
| *OPEN | 050 | SETMI | 415 | TDNN | 616 |
| OR | 434 | SETMM | 416 | TDO | 670 |
| ORB | 437 | SETO | 474 | TDOA | 674 |
| ORCA | 454 | SETOB | 477 | TDOE | 672 |
| ORCAB | 457 | SETOI | 475 | TDON | 676 |

| | | | | | |
|---|---|---|---|---|---|
| TDZ | 630 | TRCA | 644 | TSO | 671 |
| TDZA | 634 | TRCE | 642 | TSOA | 675 |
| TDZE | 632 | TRCN | 646 | TSOE | 673 |
| TDZN | 636 | TRN | 600 | TSON | 677 |
| TLC | 641 | TRNA | 604 | TSZ | 631 |
| TLCA | 645 | TRNE | 602 | TSZA | 635 |
| TLCE | 643 | TRNN | 606 | TSZE | 633 |
| TLCN | 647 | TRO | 660 | TSZN | 637 |
| TLN | 601 | TROA | 664 | *TTCALL | 051 |
| TLNA | 605 | TROE | 662 | UFA | 130 |
| TLNE | 603 | TRON | 666 | *UGETF | 073 |
| TLNN | 607 | TRZ | 620 | *UJEN | 100 |
| TLO | 661 | TRZA | 624 | *USETI | 074 |
| TLOA | 665 | TRZE | 622 | *USETO | 075 |
| TLOE | 663 | TRZN | 626 | †UTC | 210 |
| TLON | 667 | TSC | 651 | †UTS | 214 |
| TLZ | 621 | TSCA | 655 | XCT | 256 |
| TLZA | 625 | TSCE | 653 | XOR | 430 |
| TLZE | 623 | TSCN | 657 | XORB | 433 |
| TLZN | 627 | TSN | 611 | XORI | 431 |
| †TMC | 340 | TSNA | 615 | XORM | 432 |
| †TMS | 344 | TSNE | 613 | | |
| TRC | 640 | TSNN | 617 | | |

| FIRST OCTAL DIGIT \ SECOND AND THIRD OCTAL DIGITS | 00 | 04 | 10 | 14 | 20 | 24 | 30 | 34 | 40 | 44 | 50 | 54 | 60 | 64 | 70 | 74 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 6,10 APR CPA CENTRAL PROCESSOR | 6,10 PI PRIORITY INTERRUPT | 10 PAG* KI10 PAGING | 10 DA10 CCI PDP-8,9 INTERFACE | 10 DA10 CCI2 PDP-8,9 INTERFACE | 10 AD10 ADC ANALOG-DIGITAL CONVERTER | 10 AD10 ADC2 ANALOG-DIGITAL CONVERTER | | | | | | 10 DLB PDP-11 DATA LINK | DL10 DLC | 10 DK10 CLK REAL TIME CLOCK | 10 DK10 CLK2 REAL TIME CLOCK |
| 1 | 6 761 10 PTP PAPER TAPE PUNCH | 6 760 10 PTR PAPER TAPE READER | 10 CP10 CDP CARD PUNCH | 6 461 CDR CARD READER | 6 626 10 TTY CONSOLE TELETYPE | 6 646 10 LP10 LPT LINE PRINTER | 6,10 340 10 VP10 DIS DISPLAY | 6,10 340 10 VP10 DIS2 DISPLAY | 10 XY10 PLT PLOTTER | 10 XY10 PLT2 PLOTTER | 10 CR10 CR CARD READER | 10 CR10 CR2 CARD READER | 10 DLB2† PDP-11 DATA LINK | DL10 DLC2 | 10 RC10 DSK SMALL DISK | 10 RC10 DSK2 SMALL DISK |
| 2 | 6 136 DC DATA CONTROL | 6 136 DC2 DATA CONTROL | 6 UTC DECTAPE | 551 UTS | 6 MTC MAGNETIC TAPE | MTS | 516 MTM | 6 646 10 LP10 LPT2 LINE PRINTER | 10 DC10 DLS DATA LINE SCANNER | 10 DC10 DLS2 DATA LINE SCANNER | 10 RP10 DPC DISK PACK SYSTEM | 10 RP10 DPC2 DISK PACK SYSTEM | 10 RP10 DPC3 DISK PACK SYSTEM | 10 RP10 DPC4 DISK PACK SYSTEM | 6 270 DF DISK FILE | |
| 3 | 6 DCSA DATA COMMUNICATION | 630 DCSB | | | 10 DTC DECTAPE | TD10 DTS | 10 DTC2 DECTAPE | TD10 DTS2 | 10 TMC MAGNETIC TAPE | TM10 TMS | 10 TMC2 MAGNETIC TAPE | TM10 TMS2 | | | | |
| 4 | | | | | | | | | | | | | 10 DSS SINGLE SYNCHRONOUS LINE UNIT | DS10 DSI | 10 DSS2 SINGLE SYNCHRONOUS LINE UNIT | DS10 DSI2 |
| 5 | | | | | | | | | | | | | | | | |
| 6 | | | | | | | | | | | | | | | | |
| 7 | | | | | | | | | | | | | | | | |

CODES IN THIS SECTION RESERVED FOR USER SPECIAL DEVICES

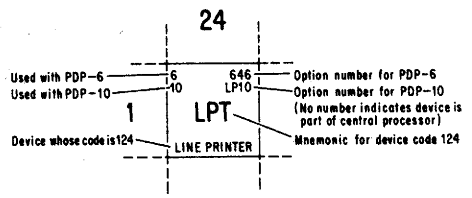
KI10 UNRESTRICTED CODES RESERVED FOR USERS

KI10 UNRESTRICTED CODES RESERVED FOR DEC

*DRUM PROCESSOR IN PDP-6

†PDP-7,8 INTERFACE IN PDP-6

| | | | | DEVICE CODE | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IN-OUT INSTRUCTION WORD | 1 | 1 | 1 | FIRST OCTAL DIGIT | | | SECOND OCTAL DIGIT | | | THIRD OCTAL DIGIT (MSB ONLY) | INSTRUCTION CODE | | |
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |



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.

| | | | |
|---|---|---|---|
| Boolean | A13 | In-out | A17 |
| Byte manipulation | A14 | Program control | A17 |
| Fixed point arithmetic | A14 | Pushdown list | A17 |
| Floating point arithmetic | A14 | Shift and rotate | A17 |
| Full word data transmission | A15 | Test, arithmetic | A18 |
| Half word data transmission | A16 | Test, logical | A19 |

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. |
| $\wedge \vee \veebar \sim$ | The Boolean operators AND, inclusive OR, exclusive OR, and complement (logical negation). |

+ − × ÷ || 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

$$(AC) + 1 \rightarrow (AC)$$
$$\textit{If}\ (AC) = 0\text{:}\quad E \rightarrow (PC)$$

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*]<br>*If* P − S ⩾ 0: P − S → P<br>*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) X (E) → (AC)* | MUL | 224 | (AC) X (E) → (AC,AC+1) |
| IMULI | 221 | (AC) X 0,E → (AC)* | MULI | 225 | (AC) X 0,E → (AC,AC+1) |
| IMULM | 222 | (AC) X (E) → (E)* | MULM | 226 | (AC) X (E) → (E)† |
| IMULB | 223 | (AC) X (E) → (AC) (E)* | MULB | 227 | (AC) X (E) → (AC,AC+1) (E) |
| IDIV | 230 | (AC) ÷ (E) → (AC)<br>REMAINDER → (AC+1) | DIV | 234 | (AC,AC+1) ÷ (E) → (AC)<br>REMAINDER → (AC+1) |
| IDIVI | 231 | (AC) ÷ 0,E → (AC)<br>REMAINDER → (AC+1) | DIVI | 235 | (AC,AC+1) ÷ 0,E → (AC)<br>REMAINDER → (AC+1) |
| IDIVM | 232 | (AC) ÷ (E) → (E) | DIVM | 236 | (AC,AC+1) ÷ (E) → (E) |
| IDIVB | 233 | (AC) ÷ (E) → (AC) (E)<br>REMAINDER → (AC+1) | DIVB | 237 | (AC,AC+1) ÷ (E) → (AC) (E)<br>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 | $(AC) \times (E) \rightarrow (AC)$ | FMPR | 164 | $(AC) \times (E) \rightarrow (AC)$ |
| FMPL | 161 | $(AC) \times (E) \rightarrow (AC,AC+1)$ | FMPRI | 165 | $(AC) \times E,0 \rightarrow (AC)$ |
| FMPM | 162 | $(AC) \times (E) \rightarrow (E)$ | FMPRM | 166 | $(AC) \times (E) \rightarrow (E)$ |
| FMPB | 163 | $(AC) \times (E) \rightarrow (AC)\ (E)$ | FMPRB | 167 | $(AC) \times (E) \rightarrow (AC)\ (E)$ |
| FDV | 170 | $(AC) \div (E) \rightarrow (AC)$ | FDVR | 174 | $(AC) \div (E) \rightarrow (AC)$ |
| FDVL | 171 | $(AC) \div (E) \rightarrow (AC)$ REMAINDER $\rightarrow (AC+1)$ | FDVRI | 175 | $(AC) \div E,0 \rightarrow (AC)$ |
| FDVM | 172 | $(AC) \div (E) \rightarrow (E)$ | FDVRM | 176 | $(AC) \div (E) \rightarrow (E)$ |
| FDVB | 173 | $(AC) \div (E) \rightarrow (AC)\ (E)$ | FDVRB | 177 | $(AC) \div (E) \rightarrow (AC)\ (E)$ |

| | | |
|---|---|---|
| UFA | 130 | $(AC) + (E) \rightarrow (AC+1)$ *without normalization* |
| DFN | 131 | $-(AC,E) \rightarrow (AC,E)$ |
| FSC | 132 | $(AC) \times 2^E \rightarrow (AC)$ |
| FLTR | 127 | (E) *floated, rounded* $\rightarrow$ (AC) |

| | | | | | |
|---|---|---|---|---|---|
| FIX | 122 | (E) *fixed* $\rightarrow$ (AC) | FIXR | 126 | (E) *fixed, rounded* $\rightarrow$ (AC) |

| | | |
|---|---|---|
| DFAD | 110 | $(AC,AC+1) + (E,E+1) \rightarrow (AC,AC+1)$ |
| DFSB | 111 | $(AC,AC+1) - (E,E+1) \rightarrow (AC,AC+1)$ |
| DFMP | 112 | $(AC,AC+1) \times (E,E+1) \rightarrow (AC,AC+1)$ |
| DFDV | 113 | $(AC,AC+1) \div (E,E+1) \rightarrow (AC,AC+1)$ |

| | | | | | |
|---|---|---|---|---|---|
| DMOVE | 120 | $(E,E+1) \rightarrow (AC,AC+1)$ | DMOVEM | 124 | $(AC,AC+1) \rightarrow (E,E+1)$ |
| DMOVN | 121 | $-(E,E+1) \rightarrow (AC,AC+1)$ | DMOVNM | 125 | $-(AC,AC+1) \rightarrow (E,E+1)$ |

## Full Word Data Transmission

| | | |
|---|---|---|
| EXCH | 250 | $(AC) \leftrightarrow (E)$ |
| BLT | 251 | *Move* $E - (AC)_R + 1$ *words starting with* $((AC)_L) \rightarrow ((AC)_R)$ [*see page 2-10*] |

| | | | | | |
|---|---|---|---|---|---|
| MOVE | 200 | $(E) \rightarrow (AC)$ | MOVS | 204 | $(E)_S \rightarrow (AC)$ |
| MOVEI | 201 | $0,E \rightarrow (AC)$ | MOVSI | 205 | $E,0 \rightarrow (AC)$ |
| MOVEM | 202 | $(AC) \rightarrow (E)$ | MOVSM | 206 | $(AC)_S \rightarrow (E)$ |
| MOVES | 203 | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | MOVSS | 207 | $(E)_S \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| MOVN | 210 | $-(E) \rightarrow (AC)$ | MOVM | 214 | $\lvert(E)\rvert \rightarrow (AC)$ |
| MOVNI | 211 | $-[0,E] \rightarrow (AC)$ | MOVMI | 215 | $0,E \rightarrow (AC)$ |
| MOVNM | 212 | $-(AC) \rightarrow (E)$ | MOVMM | 216 | $\lvert(AC)\rvert \rightarrow (E)$ |
| MOVNS | 213 | $-(E) \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | MOVMS | 217 | $\lvert(E)\rvert \rightarrow (E)$ *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |

### 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* $AC \neq 0$: $(E) \rightarrow (AC)$ | HLLZS | 513 | $0 \rightarrow (E)_R$ |
| | | | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| 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 | $777777 \rightarrow (E)_R$ | HLLES | 533 | $(E)_0 \times 777777 \rightarrow (E)_R$ |
| | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| 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 | $(E)_L \rightarrow (E)_R$ | HLRZS | 557 | $0,(E)_L \rightarrow (E)$ |
| | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| 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 | $777777,(E)_L \rightarrow (E)$ | HLRES | 577 | $[(E)_0 \times 777777],(E)_L \rightarrow (E)$ |
| | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| 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* $AC \neq 0$: $(E) \rightarrow (AC)$ | HRRZS | 553 | $0 \rightarrow (E)_L$ |
| | | | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| 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 | $777777 \rightarrow (E)_L$ | HRRES | 573 | $(E)_{18} \times 777777 \rightarrow (E)_L$ |
| | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| 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 | $(E)_R \rightarrow (E)_L$ | HRLZS | 517 | $(E)_R,0 \rightarrow (E)$ |
| | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ | | | *If* $AC \neq 0$: $(E) \rightarrow (AC)$ |

| | | | | | |
|---|---|---|---|---|---|
| HRLO | 524 | $(E)_R$,777777 → (AC) | HRLE | 534 | $(E)_R$,[$(E)_{18}$ × 777777] → (AC) |
| HRLOI | 525 | E,777777 → (AC) | HRLEI | 535 | E,[$E_{18}$ × 777777] → (AC) |
| HRLOM | 526 | $(AC)_R$,777777 → (E) | HRLEM | 536 | $(AC)_R$,[$(AC)_{18}$ × 777777] → (E) |
| HRLOS | 527 | $(E)_R$,777777 → (E) *If* AC ≠ 0: (E) → (AC) | HRLES | 537 | $(E)_R$,[$(E)_{18}$ × 777777] → (E) *If* AC ≠ 0: (E) → (AC) |

## In-out

| | | | | | |
|---|---|---|---|---|---|
| CONO | 70020 | E → COMMAND | CONSZ | 70030 | *If* $\text{STATUS}_R$ ∧ E = 0: *skip* |
| CONI | 70024 | STATUS → (E) | CONSO | 70034 | *If* $\text{STATUS}_R$ ∧ E ≠ 0: *skip* |
| DATAO | 70014 | (E) → DATA | DATAI | 70004 | DATA → (E) |

| | | | |
|---|---|---|---|
| BLKO | 70010 | (E) + 1000001 → (E)* | $((E)_R)$ → DATA [*see page 2-77*] |
| BLKI | 70000 | (E) + 1000001 → (E)* | DATA → $((E)_R)$ [*see page 2-77*] |

## 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) × $2^E$ → (AC) | ASHC | 245 | (AC,AC+1) × $2^E$ → (AC,AC+1) |
| ROT | 241 | *Rotate* (AC) E *places* | ROTC | 246 | *Rotate* (AC,AC+1) E *places* |
| LSH | 242 | *Shift* (AC) E *places* | LSHC | 247 | *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 $\rightarrow$ (AC)* | *If* (AC) $\geqslant$ 0: E $\rightarrow$ (PC) |
| AOBJN | 253 | (AC) + 1000001 $\rightarrow$ (AC)* | *If* (AC) $<$ 0: E $\rightarrow$ (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) $\leqslant$ E: *skip* | CAMLE | 313 | *If* (AC) $\leqslant$ (E): *skip* |
| CAIA | 304 | *Skip* | CAMA | 314 | *Skip* |
| CAIGE | 305 | *If* (AC) $\geqslant$ E: *skip* | CAMGE | 315 | *If* (AC) $\geqslant$ (E): *skip* |
| CAIN | 306 | *If* (AC) $\neq$ E: *skip* | CAMN | 316 | *If* (AC) $\neq$ (E): *skip* |
| CAIG | 307 | *If* (AC) $>$ E: *skip* | CAMG | 317 | *If* (AC) $>$ (E): *skip* |
| JUMP | 320 | *No-op* | SKIP | 330 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC) |
| JUMPL | 321 | *If* (AC) $<$ 0: E $\rightarrow$ (PC) | SKIPL | 331 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*If* (E) $<$ 0: *skip* |
| JUMPE | 322 | *If* (AC) = 0: E $\rightarrow$ (PC) | SKIPE | 332 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*If* (E) = 0: *skip* |
| JUMPLE | 323 | *If* (AC) $\leqslant$ 0: E $\rightarrow$ (PC) | SKIPLE | 333 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*If* (E) $\leqslant$ 0: *skip* |
| JUMPA | 324 | E $\rightarrow$ (PC) | SKIPA | 334 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*Skip* |
| JUMPGE | 325 | *If* (AC) $\geqslant$ 0: E $\rightarrow$ (PC) | SKIPGE | 335 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*If* (E) $\geqslant$ 0: *skip* |
| JUMPN | 326 | *If* (AC) $\neq$ 0: E $\rightarrow$ (PC) | SKIPN | 336 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*If* (E) $\neq$ 0: *skip* |
| JUMPG | 327 | *If* (AC) $>$ 0: E $\rightarrow$ (PC) | SKIPG | 337 | *If* AC $\neq$ 0: (E) $\rightarrow$ (AC)<br>*If* (E) $>$ 0: *skip* |
| AOJ | 340 | (AC) + 1 $\rightarrow$ (AC) | SOJ | 360 | (AC) − 1 $\rightarrow$ (AC) |
| AOJL | 341 | (AC) + 1 $\rightarrow$ (AC)<br>*If* (AC) $<$ 0: E $\rightarrow$ (PC) | SOJL | 361 | (AC) − 1 $\rightarrow$ (AC)<br>*If* (AC) $<$ 0: E $\rightarrow$ (PC) |
| AOJE | 342 | (AC) + 1 $\rightarrow$ (AC)<br>*If* (AC) = 0: E $\rightarrow$ (PC) | SOJE | 362 | (AC) − 1 $\rightarrow$ (AC)<br>*If* (AC) = 0: E $\rightarrow$ (PC) |
| AOJLE | 343 | (AC) + 1 $\rightarrow$ (AC)<br>*If* (AC) $\leqslant$ 0: E $\rightarrow$ (PC) | SOJLE | 363 | (AC) − 1 $\rightarrow$ (AC)<br>*If* (AC) $\leqslant$ 0: E $\rightarrow$ (PC) |
| AOJA | 344 | (AC) + 1 $\rightarrow$ (AC)<br>E $\rightarrow$ (PC) | SOJA | 364 | (AC) − 1 $\rightarrow$ (AC)<br>E $\rightarrow$ (PC) |
| AOJGE | 345 | (AC) + 1 $\rightarrow$ (AC)<br>*If* (AC) $\geqslant$ 0: E $\rightarrow$ (PC) | SOJGE | 365 | (AC) − 1 $\rightarrow$ (AC)<br>*If* (AC) $\geqslant$ 0: E $\rightarrow$ (PC) |

*In the KI10, 1 is added to or subtracted from each half separately.

| | | | | | |
|---|---|---|---|---|---|
| AOJN | 346 | $(AC) + 1 \rightarrow (AC)$<br>*If* $(AC) \neq 0$: $E \rightarrow (PC)$ | SOJN | 366 | $(AC) - 1 \rightarrow (AC)$<br>*If* $(AC) \neq 0$: $E \rightarrow (PC)$ |
| AOJG | 347 | $(AC) + 1 \rightarrow (AC)$<br>*If* $(AC) > 0$: $E \rightarrow (PC)$ | SOJG | 367 | $(AC) - 1 \rightarrow (AC)$<br>*If* $(AC) > 0$: $E \rightarrow (PC)$ |
| AOS | 350 | $(E) + 1 \rightarrow (E)$<br>*If* $(AC) \neq 0$: $(E) \rightarrow (AC)$ | SOS | 370 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$ |
| AOSL | 351 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) < 0$: *skip* | SOSL | 371 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) < 0$: *skip* |
| AOSE | 352 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) = 0$: *skip* | SOSE | 372 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) = 0$: *skip* |
| AOSLE | 353 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) \leqslant 0$: *skip* | SOSLE | 373 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) \leqslant 0$: *skip* |
| AOSA | 354 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*Skip* | SOSA | 374 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*Skip* |
| AOSGE | 355 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) \geqslant 0$: *skip* | SOSGE | 375 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) \geqslant 0$: *skip* |
| AOSN | 356 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) \neq 0$: *skip* | SOSN | 376 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) \neq 0$: *skip* |
| AOSG | 357 | $(E) + 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*If* $(E) > 0$: *skip* | SOSG | 377 | $(E) - 1 \rightarrow (E)$<br>*If* $AC \neq 0$: $(E) \rightarrow (AC)$<br>*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*<br>$(AC)_L \wedge \sim E \rightarrow (AC)_L$ | TRZE | 622 | *If* $(AC)_R \wedge E = 0$: *skip*<br>$(AC)_R \wedge \sim E \rightarrow (AC)_R$ |
| 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*<br>$(AC)_L \wedge \sim E \rightarrow (AC)_L$ | TRZN | 626 | *If* $(AC)_R \wedge E \neq 0$: *skip*<br>$(AC)_R \wedge \sim E \rightarrow (AC)_R$ |

## MNEMONICS

| | | | | | |
|---|---|---|---|---|---|
| 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*<br>$(AC)_L \forall E \rightarrow (AC)_L$ | TRCE | 642 | *If* $(AC)_R \wedge E = 0$: *skip*<br>$(AC)_R \forall E \rightarrow (AC)_R$ |
| 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*<br>$(AC)_L \forall E \rightarrow (AC)_L$ | TRCN | 646 | *If* $(AC)_R \wedge E \neq 0$: *skip*<br>$(AC)_R \forall E \rightarrow (AC)_R$ |
| 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*<br>$(AC)_L \vee E \rightarrow (AC)_L$ | TROE | 662 | *If* $(AC)_R \wedge E = 0$: *skip*<br>$(AC)_R \vee E \rightarrow (AC)_R$ |
| 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*<br>$(AC)_L \vee E \rightarrow (AC)_L$ | TRON | 666 | *If* $(AC)_R \wedge E \neq 0$: *skip*<br>$(AC)_R \vee E \rightarrow (AC)_R$ |
| 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*<br>$(AC) \wedge \sim (E) \rightarrow (AC)$ | TSZE | 633 | *If* $(AC) \wedge (E)_S = 0$: *skip*<br>$(AC) \wedge \sim (E)_S \rightarrow (AC)$ |
| 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*<br>$(AC) \wedge \sim (E) \rightarrow (AC)$ | TSZN | 637 | *If* $(AC) \wedge (E)_S \neq 0$: *skip*<br>$(AC) \wedge \sim (E)_S \rightarrow (AC)$ |
| TDC | 650 | $(AC) \forall (E) \rightarrow (AC)$ | TSC | 651 | $(AC) \forall (E)_S \rightarrow (AC)$ |
| TDCE | 652 | *If* $(AC) \wedge (E) = 0$: *skip*<br>$(AC) \forall (E) \rightarrow (AC)$ | TSCE | 653 | *If* $(AC) \wedge (E)_S = 0$: *skip*<br>$(AC) \forall (E)_S \rightarrow (AC)$ |
| 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*<br>$(AC) \forall (E) \rightarrow (AC)$ | TSCN | 657 | *If* $(AC) \wedge (E)_S \neq 0$: *skip*<br>$(AC) \forall (E)_S \rightarrow (AC)$ |
| TDO | 670 | $(AC) \vee (E) \rightarrow (AC)$ | TSO | 671 | $(AC) \vee (E)_S \rightarrow (AC)$ |
| TDOE | 672 | *If* $(AC) \wedge (E) = 0$: *skip*<br>$(AC) \vee (E) \rightarrow (AC)$ | TSOE | 673 | *If* $(AC) \wedge (E)_S = 0$: *skip*<br>$(AC) \vee (E)_S \rightarrow (AC)$ |
| 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*<br>$(AC) \vee (E) \rightarrow (AC)$ | TSON | 677 | *If* $(AC) \wedge (E)_S \neq 0$: *skip*<br>$(AC) \vee (E)_S \rightarrow (AC)$ |

## APPENDIX B

## INPUT-OUTPUT CODES

The table beginning on the next page lists the complete teletype code. The lower case character set (codes 140–176) is not available on the Model 35, but giving one of these codes causes the teletype to print the corresponding upper case character. Other differences between the 35 and 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the console teletype, and the definitions bear no necessary relation to the use of the codes in conjunction with the DECsystem–10 software.

The line printer has the same codes and characters as the teletype. The 64-character printer has the figure and upper case sets, codes 040–137 (again, giving a lower case code prints the upper case character). The "96"-character printer has these plus the lower case set, codes 040–176. The latter printer actually has only ninety-five characters unless a special character is "hidden" under the delete code, 177. A hidden character is printed by sending its code prefixed by the delete code. Hence a character hidden under DEL is printed by sending the printer two 177s in a row.

Besides printing characters, the line printer responds to ten control characters, HT, CR, LF, VT, FF, DLE and DC1–4. The 128-character printer uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control characters that affect the printer and also under null and delete. In all cases, prefixing DEL causes the hidden character to be printed. The extra thirty-three characters that complete the set are ordered special for each installation.

The first page of the table of card codes [*pages B6–8*] lists the column punch required to represent any character in the two DEC codes. The octal codes listed are those used by the DECsystem–10 software. In other words, when reading cards, the Monitor translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. The remaining pages of the table show the relationship between the DEC card codes and several IBM card punches. Each of the column punches is produced by a single key on any punch for which a character is listed, the character being that which is printed at the top of the card.

## TELETYPE CODE

| Even Parity Bit | 7-Bit Octal Code | Character | Remarks |
|---|---|---|---|
| 0 | 000 | NUL | Null, tape feed. Repeats on Model 37. Control shift P on Model 35. |
| 1 | 001 | SOH | Start of heading; also SOM, start of message. Control A. |
| 1 | 002 | STX | Start of text; also EOA, end of address. Control B. |
| 0 | 003 | ETX | End of text; also EOM, end of message. Control C. |
| 1 | 004 | EOT | End of transmission (END); shuts off TWX machines. Control D. |
| 0 | 005 | ENQ | Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is . . . ") at remote station if so equipped. Control E. |
| 0 | 006 | ACK | Acknowledge; also RU, "Are you . . . ?" Control F. |
| 1 | 007 | BEL | Rings the bell. Control G. |
| 1 | 010 | BS | Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 35. |
| 0 | 011 | HT | Horizontal tab. Control I on Model 35. |
| 0 | 012 | LF | Line feed or line space (NEW LINE); advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 35. |
| 1 | 013 | VT | Vertical tab (VTAB). Control K on Model 35. |
| 0 | 014 | FF | Form feed to top of next page (PAGE). Control L. |
| 1 | 015 | CR | Carriage return to beginning of line. Control M on Model 35. |
| 1 | 016 | SO | Shift out; changes ribbon color to red. Control N. |
| 0 | 017 | SI | Shift in; changes ribbon color to black. Control O. |
| 1 | 020 | DLE | Data link escape. Control P (DC0). |
| 0 | 021 | DC1 | Device control 1, turns transmitter (reader) on. Control Q (X ON). |
| 0 | 022 | DC2 | Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON). |
| 1 | 023 | DC3 | Device control 3, turns transmitter (reader) off. Control S (X OFF). |
| 0 | 024 | DC4 | Device control 4, turns punch or auxiliary off. Control T (~~TAPE~~, AUX OFF). |
| 1 | 025 | NAK | Negative acknowledge; also ERR, error. Controi U. |
| 1 | 026 | SYN | Synchronous idle (SYNC). Control V. |
| 0 | 027 | ETB | End of transmission block; also LEM, logical end of medium. Control W. |
| 0 | 030 | CAN | Cancel (CANCL). Control X. |
| 1 | 031 | EM | End of medium. Control Y. |
| 1 | 032 | SUB | Substitute. Control Z. |
| 0 | 033 | ESC | Escape, prefix. This code is generated by control shift K on Model 35, but the Monitor translates it to 175. |
| 1 | 034 | FS | File separator. Control shift L on Model 35. |
| 0 | 035 | GS | Group separator. Control shift M on Model 35. |

TELETYPE CODE B3

| Even Parity Bit | 7-Bit Octal Code | Character | Remarks |
|---|---|---|---|
| 0 | 036 | RS | Record separator. Control shift N on Model 35. |
| 1 | 037 | US | Unit separator. Control shift O on Model 35. |
| 1 | 040 | SP | Space. |
| 0 | 041 | ! | |
| 0 | 042 | " | |
| 1 | 043 | # | |
| 0 | 044 | $ | |
| 1 | 045 | % | |
| 1 | 046 | & | |
| 0 | 047 | ' | Accent acute or apostrophe. |
| 0 | 050 | ( | |
| 1 | 051 | ) | |
| 1 | 052 | * | Repeats on Model 37. |
| 0 | 053 | + | |
| 1 | 054 | , | |
| 0 | 055 | - | Repeats on Model 37. |
| 0 | 056 | . | Repeats on Model 37. |
| 1 | 057 | / | |
| 0 | 060 | Ø | |
| 1 | 061 | 1 | |
| 1 | 062 | 2 | |
| 0 | 063 | 3 | |
| 1 | 064 | 4 | |
| 0 | 065 | 5 | |
| 0 | 066 | 6 | |
| 1 | 067 | 7 | |
| 1 | 070 | 8 | |
| 0 | 071 | 9 | |
| 0 | 072 | : | |
| 1 | 073 | ; | |
| 0 | 074 | < | |
| 1 | 075 | = | Repeats on Model 37. |
| 1 | 076 | > | |
| 0 | 077 | ? | |
| 1 | 100 | @ | |
| 0 | 101 | A | |
| 0 | 102 | B | |

B4 INPUT-OUTPUT CODES

| Even Parity Bit | 7-Bit Octal Code | Character | Remarks |
|---|---|---|---|
| 1 | 103 | C | |
| 0 | 104 | D | |
| 1 | 105 | E | |
| 1 | 106 | F | |
| 0 | 107 | G | |
| 0 | 110 | H | |
| 1 | 111 | I | |
| 1 | 112 | J | |
| 0 | 113 | K | |
| 1 | 114 | L | |
| 0 | 115 | M | |
| 0 | 116 | N | |
| 1 | 117 | O | |
| 0 | 120 | P | |
| 1 | 121 | Q | |
| 1 | 122 | R | |
| 0 | 123 | S | |
| 1 | 124 | T | |
| 0 | 125 | U | |
| 0 | 126 | V | |
| 1 | 127 | W | |
| 1 | 130 | X | Repeats on Model 37. |
| 0 | 131 | Y | |
| 0 | 132 | Z | |
| 1 | 133 | [ | Shift K on Model 35. |
| 0 | 134 | \ | Shift L on Model 35. |
| 1 | 135 | ] | Shift M on Model 35. |
| 1 | 136 | ↑ | |
| 0 | 137 | ← | Repeats on Model 37. |
| 0 | 140 | ` | Accent grave. |
| 1 | 141 | a | |
| 1 | 142 | b | |
| 0 | 143 | c | |
| 1 | 144 | d | |
| 0 | 145 | e | |
| 0 | 146 | f | |
| 1 | 147 | g | |

| Even Parity Bit | 7-Bit Octal Code | Character | Remarks |
|---|---|---|---|
| 1 | 150 | h | |
| 0 | 151 | i | |
| 0 | 152 | j | |
| 1 | 153 | k | |
| 0 | 154 | l | |
| 1 | 155 | m | |
| 1 | 156 | n | |
| 0 | 157 | o | |
| 1 | 160 | p | |
| 0 | 161 | q | |
| 0 | 162 | r | |
| 1 | 163 | s | |
| 0 | 164 | t | |
| 1 | 165 | u | |
| 1 | 166 | v | |
| 0 | 167 | w | |
| 0 | 170 | x | Repeats on Model 37. |
| 1 | 171 | y | |
| 1 | 172 | z | |
| 0 | 173 | { | |
| 1 | 174 | \| | |
| 0 | 175 | } | This code generated by ALT MODE on Model 35. |
| 0 | 176 | ~ | This code generated by ESC key (if present) on Model 35, but the Monitor translates it to 175. |
| 1 | 177 | DEL | Delete, rub out. Repeats on Model 37. |

**Keys That Generate No Codes**

| | |
|---|---|
| REPT | Model 35 only: causes any other key that is struck to repeat continuously until REPT is released. |
| PAPER ADVANCE | Model 37 local line feed. |
| LOCAL RETURN | Model 37 local carriage return. |
| LOC LF | Model 35 local line feed. |
| LOC CR | Model 35 local carriage return. |
| INTERRUPT, BREAK | Opens the line (machine sends a continuous string of null characters). |
| PROCEED, BRK RLS | Break release (not applicable). |
| HERE IS | Transmits predetermined 21-character message. |

## CARD CODES

| Character | PDP-10 ASCII | DEC 029 | DEC 026 | Character | PDP-10 ASCII | DEC 029 | DEC 026 |
|---|---|---|---|---|---|---|---|
| *Space* | 040 | *None* | *None* | @ | 100 | 8 4 | 8 4 |
| ! | 041 | 11 8 2 | 12 8 7 | A | 101 | 12 1 | 12 1 |
| " | 042 | 8 7 | 0 8 5 | B | 102 | 12 2 | 12 2 |
| # | 043 | 8 3 | 0 8 6 | C | 103 | 12 3 | 12 3 |
| $ | 044 | 11 8 3 | 11 8 3 | D | 104 | 12 4 | 12 4 |
| % | 045 | 0 8 4 | 0 8 7 | E | 105 | 12 5 | 12 5 |
| & | 046 | 12 | 11 8 7 | F | 106 | 12 6 | 12 6 |
| ' | 047 | 8 5 | 8 6 | G | 107 | 12 7 | 12 7 |
| ( | 050 | 12 8 5 | 0 8 4 | H | 110 | 12 8 | 12 8 |
| ) | 051 | 11 8 5 | 12 8 4 | I | 111 | 12 9 | 12 9 |
| * | 052 | 11 8 4 | 11 8 4 | J | 112 | 11 1 | 11 1 |
| + | 053 | 12 8 6 | 12 | K | 113 | 11 2 | 11 2 |
| , | 054 | 0 8 3 | 0 8 3 | L | 114 | 11 3 | 11 3 |
| − | 055 | 11 | 11 | M | 115 | 11 4 | 11 4 |
| . | 056 | 12 8 3 | 12 8 3 | N | 116 | 11 5 | 11 5 |
| / | 057 | 0 1 | 0 1 | O | 117 | 11 6 | 11 6 |
| 0 | 060 | 0 | 0 | P | 120 | 11 7 | 11 7 |
| 1 | 061 | 1 | 1 | Q | 121 | 11 8 | 11 8 |
| 2 | 062 | 2 | 2 | R | 122 | 11 9 | 11 9 |
| 3 | 063 | 3 | 3 | S | 123 | 0 2 | 0 2 |
| 4 | 064 | 4 | 4 | T | 124 | 0 3 | 0 3 |
| 5 | 065 | 5 | 5 | U | 125 | 0 4 | 0 4 |
| 6 | 066 | 6 | 6 | V | 126 | 0 5 | 0 5 |
| 7 | 067 | 7 | 7 | W | 127 | 0 6 | 0 6 |
| 8 | 070 | 8 | 8 | X | 130 | 0 7 | 0 7 |
| 9 | 071 | 9 | 9 | Y | 131 | 0 8 | 0 8 |
| : | 072 | 8 2 | 11 8 2 *or* 11 0 | Z | 132 | 0 9 | 0 9 |
| ; | 073 | 11 8 6 | 0 8 2 | [ | 133 | 12 8 2 | 11 8 5 |
| < | 074 | 12 8 4 | 12 8 6 | \ | 134 | 11 8 7 | 8 7 |
| = | 075 | 8 6 | 8 3 | ] | 135 | 0 8 2 | 12 8 5 |
| > | 076 | 0 8 6 | 11 8 6 | ↑ | 136 | 12 8 7 | 8 5 |
| ? | 077 | 0 8 7 | 12 8 2 *or* 12 0 | ← | 137 | 0 8 5 | 8 2 |

| | |
|---|---|
| *Binary* | 7 9 |
| *Mode Switch* | 12 0 2 4 6 8 |
| *End of File* | 12 11 0 1, 6 7 8 9, 12 11 0 1 6 7 8 9 |

The octal codes given above are those generated by the Monitor from the column punches. The card reader interface actually supplies a direct binary equivalent of the column punch, as listed in the following two pages.

The first end-of-file punch is not recognized by Card Reader Stacker (CDRSTK); the other two are recognized only by Card Reader Stacker.

CARD CODES

B7

| Column Punch | Character | Octal | Column Punch | Character | Octal |
|---|---|---|---|---|---|
| *None* | *Space* | 0000 | 12 9 | I | 4001 |
| 0 | 0 | 1000 | 11 1 | J | 2400 |
| 1 | 1 | 0400 | 11 2 | K | 2200 |
| 2 | 2 | 0200 | 11 3 | L | 2100 |
| 3 | 3 | 0100 | 11 4 | M | 2040 |
| 4 | 4 | 0040 | 11 5 | N | 2020 |
| 5 | 5 | 0020 | 11 6 | O | 2010 |
| 6 | 6 | 0010 | 11 7 | P | 2004 |
| 7 | 7 | 0004 | 11 8 | Q | 2002 |
| 8 | 8 | 0002 | 11 9 | R | 2001 |
| 9 | 9 | 0001 | 0 1 | / | 1400 |
| 12 1 | A | 4400 | 0 2 | S | 1200 |
| 12 2 | B | 4200 | 0 3 | T | 1100 |
| 12 3 | C | 4100 | 0 4 | U | 1040 |
| 12 4 | D | 4040 | 0 5 | V | 1020 |
| 12 5 | E | 4020 | 0 6 | W | 1010 |
| 12 6 | F | 4010 | 0 7 | X | 1004 |
| 12 7 | G | 4004 | 0 8 | Y | 1002 |
| 12 8 | H | 4002 | 0 9 | Z | 1001 |

| Column Punch | 026 Data Processing | 026 Fortran | 029 | DEC 026 | DEC 029 | Octal |
|---|---|---|---|---|---|---|
| 12 | & | + | & | + | & | 4000 |
| 11 | – | – | – | – | – | 2000 |
| 12 0 | | | | ? | | 5000 |
| 11 0 | | | | : | | 3000 |
| 8 2 | | | : | ← | : | 0202 |
| 8 3 | # | = | # | = | # | 0102 |
| 8 4 | @ | – | @ | @ | @ | 0042 |
| 8 5 | | | ' | ↑ | ' | 0022 |
| 8 6 | | | = | ' | = | 0012 |
| 8 7 | | | " | \ | " | 0006 |
| 12 8 2 | | | ¢ | ? | [ | 4202 |
| 12 8 3 | . | . | . | . | . | 4102 |
| 12 8 4 | ¤ | ) | < | ) | < | 4042 |
| 12 8 5 | | | ( | ] | ( | 4022 |
| 12 8 6 | | | + | < | + | 4012 |

| Column Punch | 026 Data Processing | 026 Fortran | 029 | DEC 026 | DEC 029 | Octal |
|---|---|---|---|---|---|---|
| 12 8 7 | | | \| | ! | ↑ | 4006 |
| 11 8 2 | | | ! | : | ! | 2202 |
| 11 8 3 | $ | $ | $ | $ | $ | 2102 |
| 11 8 4 | * | * | * | * | * | 2042 |
| 11 8 5 | | | ) | [ | ) | 2022 |
| 11 8 6 | | | ; | > | ; | 2012 |
| 11 8 7 | | | ¬ | & | \ | 2006 |
| 0 8 2 | | | *See note* | ; | ] | 1202 |
| 0 8 3 | , | , | , | , | , | 1102 |
| 0 8 4 | % | ( | % | ( | % | 1042 |
| 0 8 5 | | | ← | " | ← | 1022 |
| 0 8 6 | | | > | # | > | 1012 |
| 0 8 7 | | | ? | % | ? | 1006 |
| 12 11 0 1 | | | | *End of File** | *End of File** | 7400 |
| 12 0 2 4 6 8 | | | | *Mode Switch* | *Mode Switch* | 5252 |
| 7 9 | | | | *Binary* | *Binary* | *xx*05 |
| 6 7 8 9 | | | | *End of File*† | *End of File*† | |
| 12 11 0 1 6 7 8 9 | | | | *End of File*† | *End of File*† | |

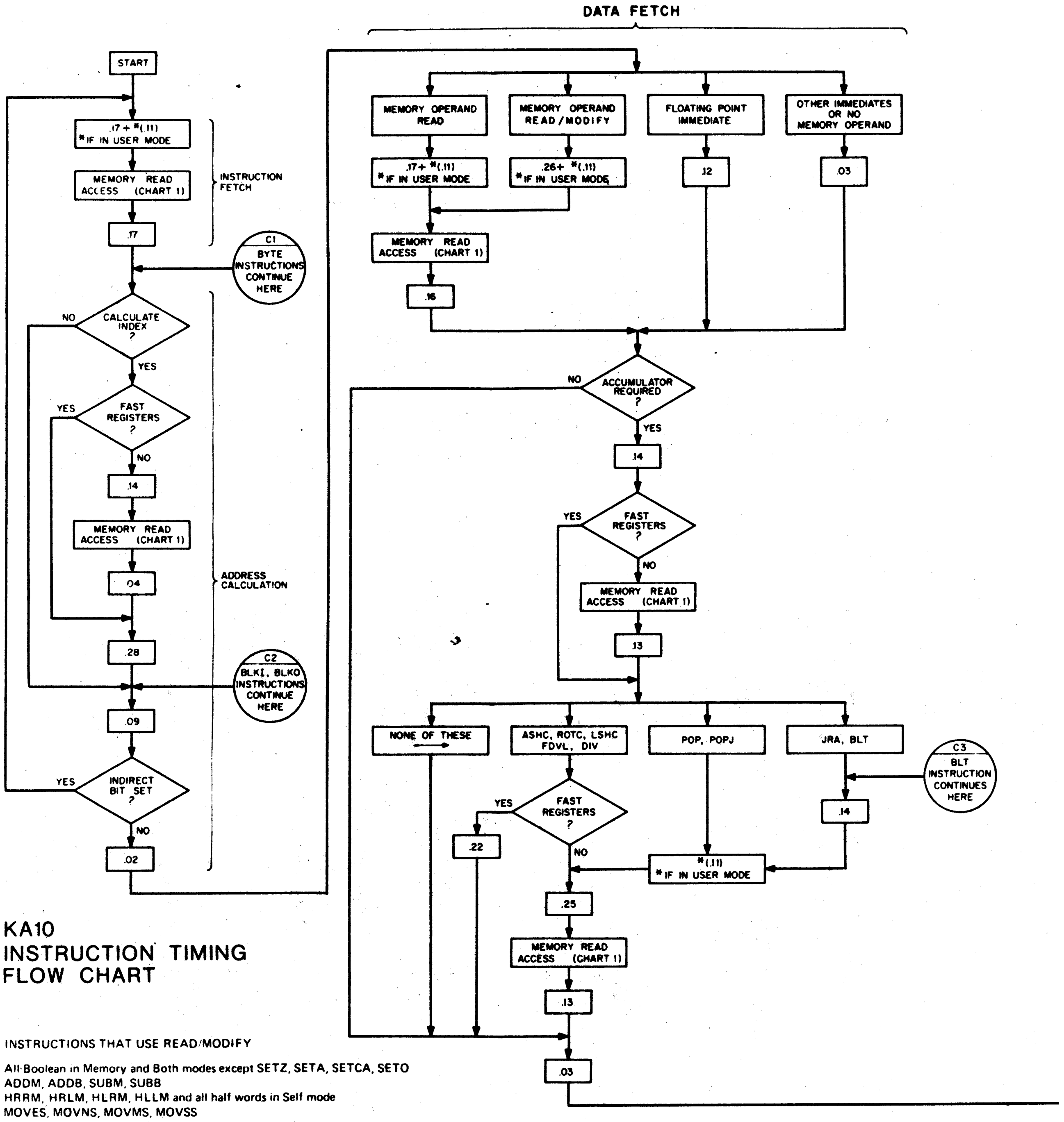
NOTE: There is a single key for the 0 8 2 punch on the 029 but printing is suppressed.
The Monitor translates the octal code for the 12 0 punch in DEC 026 to 4202 (which corresponds to a 12 8 2 punch), and the code for 11 0 to 2202 (11 8 2).

*Not recognized as end of file by Card Reader Stacker (CDRSTK).
†Recognized only by Card Reader Stacker (CDRSTK).

## APPENDIX C

## TIMING

The chart on the next two pages shows the detailed timing for the KA10. A similar chart for the KI10 and timing tables for both processors will be added later.



KA10
INSTRUCTION TIMING
FLOW CHART

INSTRUCTIONS THAT USE READ/MODIFY

All Boolean in Memory and Both modes except SETZ, SETA, SETCA, SETO
ADDM, ADDB, SUBM, SUBB
HRRM, HRLM, HLRM, HLLM and all half words in Self mode
MOVES, MOVNS, MOVMS, MOVSS
ILDB, IDPB (first time only)
IBP, BLKI, BLKO, DFN, EXCH
AOS, SOS in all modes

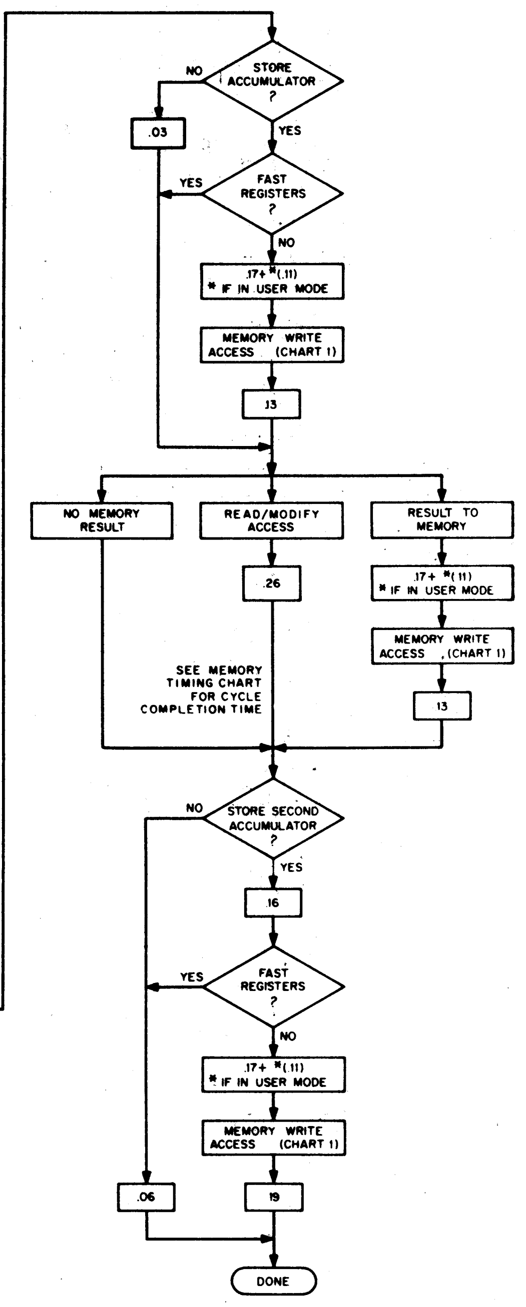
## KA10 TIMING

### INSTRUCTION EXECUTION

| | | | |
|---|---|---|---|
| Boolean (except ANDCA, ANDCB, ORCA, ORCB), Half Words (except HLR, HLRI, HRL, HRLI), MOVE, MOVS, EXCH, JFCL, JRST, JSP, XCT, UUO | | .27 | |
| ANDCA, ANDCB, ORCA, ORCB, HLR, HLRI, HRL, HRLI, JSR, JSA, JRA, Test class | | .62 | |
| MOVN, MOVM, ADD, SUB, AOBJP, AOBJN, CAM, CAI, SKIP, JUMP, AOJ, AOS, SOJ, SOS | | .45 | |
| PUSH, PUSHJ, POP, POPJ, DFN | | .80 | |
| JFFO | | .80 | + .19 times number of leading 0s mod 18 |
| BLT | | .69 | (+ .11 if User) + memory write access + .52 If not done + .09 and go to C3 |
| IBP | | .38 | + .26 if overflow word boundary |
| LDB, DPB | First time | .61 | + .15 per size count Go to C1 |
| ILDB, IDPB | First time | .74 | { + .15 per size count + .26 if overflow } Go to C1 |
| ILDB, LDB | Second time | .45 | + .15 per position count |
| IDPB, DPB | Second time | .95 | + .15 per position count |
| Shift group | | { .39 Left .23 Right } | + .15 per shift |
| MUL | | 6.02 | + .13 per transition |
| Average except MULI | | 8.36 | (18 transitions for 2.34) |
| IMUL | | 6.34 | + .13 per transition |
| Average except IMULI | | 7.51 | (9 transitions for 1.17) |
| FMP | | 6.39 | + .13 per transition |
| Average except FMPRI | | 8.21 | (14 transitions for 1.82) |
| Note: Immediate mode multiplication has only half the average number of transitions | | | |
| DIV, IDIV | | 13.78 | |
| FSC | | 1.52 | + .25 per shift to normalize |
| FAD, UFA | | 2.38 | { + .15 per shift to unnormalize + .25 per shift to normalize |
| Average | | 4.33 | |
| FSB | | Same as FAD + .18 | |
| Rounding (except divide) only when actually done | | +.96 | |
| Long mode (except divide) | | +.69 | |
| FDVR, FDV (except FDVL) | | 12.00 | |
| FDVL with fast ACs | | 13.28 | |
| FDVL without fast ACs | | 12.32 | (+ .11 if User) + memory read access + .89 |
| CONO, CONI, CONSO, CONSZ, DATAO, DATAI | | .12 | Then wait until 4.50 has passed since last here |
| CONO, CONI, DATAO, DATAI | | +2.69 | |
| CONSO, CONSZ | | +2.90 | |
| BLKO, BLKI | | .60 | Then turn into DATAO, DATAI and go to C2 |

.03

### DATA STORE


STORE ACCUMULATOR ?
NO
.03
YES
FAST REGISTERS ?
YES
NO
.17+ *(.11) * IF IN USER MODE
MEMORY WRITE ACCESS (CHART 1)
13
NO MEMORY RESULT
READ/MODIFY ACCESS
RESULT TO MEMORY
.26
.17+ *(.11) * IF IN USER MODE
MEMORY WRITE ACCESS (CHART 1)
13
SEE MEMORY TIMING CHART FOR CYCLE COMPLETION TIME
STORE SECOND ACCUMULATOR ?
NO
YES
.16
FAST REGISTERS ?
YES
NO
.17+ *(.11) * IF IN USER MODE
MEMORY WRITE ACCESS (CHART 1)
.06
.19
DONE

### MEMORY TIMING

| MEMORY | MA10 | MB10 | MB10 | FAST | MD10 | ME10 |
|---|---|---|---|---|---|---|
| PROCESSORS | SINGLE OR MULTI | SINGLE | MULTI | SINGLE (BUILT IN) | SINGLE OR MULTI | SINGLE OR MULTI |
| CYCLE | 1.00 | 1.65 | 1.75 | — | 1.8 | 1.00 |
| READ ACCESS | .61 | .60 | .70 | .21 | .83 | .61 |
| WRITE ACCESS | .20 | .20 | .30 | .21 | .33 | .20 |
| MODIFY COMPLETION | .57 | .97 | .97 | — | 1.23 | .65 |

NOTES:

MEMORY ACCESS TIMES INCLUDE DELAY INTRODUCED BY 10 FEET OF CABLE

ALL TIMES ARE NOMINAL MAXIMUMS

## APPENDIX D

## KA10 ALGORITHMS

This appendix treats only the algorithms used in the KA10; for information on the KI10 algorithms refer to the maintenance manual.

All arithmetic operations on full and half words are performed in the 36-bit parallel adder. There are two sets of summand inputs to the adder, each set of 36 supplying one input to each adder stage. One set supplies the contents of AR, its complement, or zero; the other set supplies the contents of BR, its complement, or zero. Each stage also has a carry input, which is generated by the next less significant stage. Every stage has two outputs; the carry already mentioned, and a sum. The 36 sum outputs together form the sum of the two input words. The least significant stage has a carry input from the logic for performing twos complement arithmetic and incrementing by one. The negative of a number is formed at the sum outputs simply by supplying the complement of the number at one set of inputs and asserting the carry into stage 35. Adder stage 17 has extra input gating so that 1 can be added to or subtracted from both halves of AR simultaneously.

The adder produces a sum in the same way that one adds binary numbers using pencil and paper. Each adder stage has three inputs, two summand bits and a carry, and two outputs, sum and carry. The sum output of a given stage is 1 if any one or all three of the inputs are 1. The carry out is 1 if two or three of the inputs are 1. Calculations are performed 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 arithmetic as signed numbers with 35 magnitude bits or as unsigned 36-bit numbers. A computation on signed numbers produces a result which 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.

All operations discussed below have two operands, one of which is supplied to the adder from BR, which acts simply as a buffer and has no special input gating. MQ has shift gating so it can function as a low order extension of AR for handling double length operands. All actual computations take place in the single 36-bit adder, but the sum output can be placed in either AR or MQ, and all transfers to MQ from AR or BR are made through the adder. In multiplication MQ holds the multiplier and thus controls the summation of partial products; as the multiplier is shifted out, the low order word of the product is shifted in. In division MQ supplies the low order part of the dividend to AR as the quotient is being constructed in MQ.

In any extended arithmetic operation, the requisite number of steps is counted in the 9-bit shift counter SC, which has a carry network for this purpose. SC also has a 9-bit adder for use in computations on floating point exponents and size and position calculations in byte manipulation.

## FIXED POINT ALGORITHMS

Fixed point numbers are explained in detail in § 1.1. For convenience let us take the computer representation of the positive number $x$ as $+[x]$ where the brackets enclose the number in bits 1–35. Similarly the representation of $-x$ is $-[2^{35}-x]$ or $-[1-x]$ depending on whether we are regarding numbers as integers or as proper fractions. The most negative number, $-2^{35}$, has the form $-[0]$, which is equivalent to the unsigned integer $2^{35}$.

**Addition.** There are four cases of addition of two positive 35-bit numbers $x$ and $y$.

| | | |
|---|---|---|
| I. | $x+y$ | |
| II. | $(-x)+(-y)$ | |
| III. | $x+(-y)$, | $x \geqslant y$ |
| IV. | $x+(-y)$, | $x < y$ |

The operands are held in AR and BR, but it makes no difference which one is in which register. The result appears in AR. For convenience in the exposition we shall regard the numbers as proper fractions; to view them as integers, simply substitute "$2^{35}$" for each occurrence of "1". Since the twos complement format allows a representation for $-1$, either $x$ or $y$ may be 1 in II, and $y$ may be 1 in IV.

I. If $x+y<1$ the adder output placed in AR is $+[x+y]$. If $x+y \geqslant 1$ the carry out of stage 1 changes the sign. Consequently if the addition of two positive numbers gives a negative result, it is apparent that the sum exceeds the capacity of the register. The processor detects the overflow by checking the sign carries: there is a carry into the sign stage but none out of it. AR then contains

$$-[x+y-1]$$

II. Ignoring the carry into the sign bit in the addition of two negatives would give

$$\begin{array}{l} -[1-x] \\ \underline{-[1-y]\quad\;\;} \\ +[1+1-x-y] \end{array}$$

If $x+y \leqslant 1$ the carry changes the sign and the result is

$$-[1-x-y]$$

which is the representation of $-(x+y)$. If $x+y>1$ there is no carry into the sign, and its absence in the presence of a carry out indicates overflow. AR contains

$$+[1-(x+y-1)]$$

III. Ignoring the carry into the sign in an addition where the signs are different would give

$$\begin{array}{l} +[x] \\ \underline{-[1-y]} \\ -[1+x-y] \end{array}$$

Since $x \geqslant y$, it follows that $1+x-y \geqslant 1$. Hence the carry changes the sign and the result is

$$+[x-y]$$

When the operand signs are different, the magnitude of the result cannot exceed the larger operand magnitude and there can be no overflow. Since in this case the positive number is at least as large in magnitude as the negative, there is always a carry into the sign, and this added to the operand minus sign produces a carry out.

IV. The addition of numbers of differing signs where the negative has the larger magnitude gives

$$\begin{array}{l} +[x] \\ \underline{-[1-y]} \\ -[1+x-y] \end{array}$$

Since $x<y$, then $1+x-y<1$. Hence there are no carries associated with the sign and no overflow. The above result is the twos complement representation of $x-y$, *ie* $-(y-x)$.

**Subtraction.** The minuend from AC is in AR, and the subtrahend, which is either $0,E$ or the word from location $E$, is in BR. Subtraction is done directly by adding the twos complement of BR to AR. The logic supplies the complement of BR to the adder and a carry into the adder LSB.

Let $x$ be the absolute value of the number in AR, and $y$ the absolute value of the number in BR. There are four cases.

| | | | | |
|---|---|---|---|---|
| I. | $x-(-y)$ | | | |
| II. | $(-x)-y$ | | | |
| III. | $x-y,$ | $x \geqslant y;$ | $(-x)-(-y),$ | $x \leqslant y$ |
| IV. | $x-y,$ | $x<y;$ | $(-x)-(-y),$ | $x>y$ |

These correspond respectively to the four cases of addition discussed previously.

**Multiplication.** The multiplier, $0,E$ or the contents of location $E$, is in MQ, and the multiplicand from AC is in BR. AR is clear. The 36-step procedure is as follows.

If MQ35 (the multiplier LSB) is 1, subtract BR algebraically from AR, but put the result in AR shifted one place to the right, with the LSB of the result going into MQ0, and shift MQ right so a bit of the multiplier is dropped from MQ35. Put the sign of the result in AR0 and AR1 (as though the shift followed the subtraction and did not affect the sign but did move it to AR1). If MQ35 is 0, simply shift AR and MQ right one, with AR35 going into MQ0.

In each subsequent step perform only the shift if the bits moved in and out of MQ35 on the previous step were the same. If they were different, add or subtract along with the shift: if the shift moved a 0 in and a 1 out, add BR to AR; if a 1 in and a 0 out, subtract BR from AR.

Thus the low order bits of the running sum of partial products are shifted into MQ as the multiplier is shifted out. At each step the effect of the multiplicand in BR on the partial sum in AR is one binary order of magnitude greater than in the preceding step because the partial sum was shifted right. Therefore BR can be combined directly with AR. If MQ35 is initially 0, there is no subtraction until a 1 is shifted into it. Simple shifting then continues until the next transition (from 1 to 0), following which BR is added.

The process continues in this way, subtracting at every 0-1 transition, adding at every 1-0 transition. After 35 steps MQ0–34 contains the low half of the product magnitude, and MQ35 contains the sign of the multiplier. At the final step, add or subtract as required but put the result directly into AR; shift only MQ to move the low magnitude into the correct position and make MQ0 equal to the sign of the whole product.

If the original operands were both negative and the result is also negative, set Overflow; this can occur only when $-2^{35}$ is squared. In IMUL, if the high word is not null (*ie* if AR is neither clear nor all 1s), set Overflow; move MQ to AR for storage of the low word.

To see that this procedure results in a correct product, consider the positive binary integer

$$\begin{array}{ccccccccc} 1&0&0&1&1&1&0&1&1 \\ \scriptstyle 8&\scriptstyle 7&\scriptstyle 6&\scriptstyle 5&\scriptstyle 4&\scriptstyle 3&\scriptstyle 2&\scriptstyle 1&\scriptstyle 0 \end{array}$$

where the decimal digits below the binary digits are the powers of 2 corresponding to the bit positions. This number is obviously equal to

$$\begin{array}{rr} & 100000000 \\ + & 111000 \\ + & 11 \end{array}$$

Now an $n$-bit string of 1s whose rightmost bit corresponds to $2^k$ is equal to $2^{k+n} - 2^k$, or equivalently $2^k(2^n - 2^0)$, *ie* $2^n - 2^0$ is a string of $n$ 1s and the $2^k$ shifts the string left $k$ places. Hence

$$\begin{array}{rcccl} 100000000 & = & 2^{8+1} - 2^8 & = & 2^9 - 2^8 \\ 111000 & = & 2^{3+3} - 2^3 & = & 2^6 - 2^3 \\ 11 & = & 2^{2+0} - 2^0 & = & 2^2 - 2^0 \\ \hline 100111011 & & = & & 2^9 - 2^8 + 2^6 - 2^3 + 2^2 - 2^0 \end{array}$$

In this last representation, each power of 2 that is subtracted corresponds to a transition from 0 to 1 (scanning right to left), whereas each that is added corresponds to a 1-0 transition. The largest term corresponds to the transition to the sign bit, which is 0 for a positive number. The multiplication algorithm interprets the multiplier in this manner, alternately subtracting and adding the multiplicand to the partial sum in the order-of-magnitude positions corresponding to the transitions. If a multiplier of the same magnitude were negative, it would have the form

$$\begin{array}{cccccccccc} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ - & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{array}$$

in which the extra bit at the left represents the sign. The number is now equivalent to

$$-2^9 + 2^8 - 2^6 + 2^3 - 2^2 + 2^1 - 2^0$$

wherein opposite signs correspond to opposite transitions. The algorithm may thus use exactly the same sequence for a negative multiplier: this time the subtraction of greatest magnitude is detected by the transition to the sign bit, which is now 1.

**Division.** The divisor, $0,E$ or the contents of location $E$, is in BR. In DIV the high and low halves of the dividend from two accumulators are in AR and MQ respectively. In IDIV the one-word dividend from AC is in AR. The two types of division differ mainly in setting up the dividend; in both cases the algorithm processes a positive dividend to get a positive quotient.

In DIV if the dividend is negative (AR0 = 1), make it positive and set the negative dividend flag. To negate the dividend, move the low word to AR and the complement of the high word to MQ. Then move the negative of the low word back to MQ and the complement of the high word back to AR. Now the double length negative of the original dividend is in AR and MQ unless MQ is clear; in this event add 1 to AR to give the twos complement negative of the high word. Once the dividend is in positive form shift MQ left one place to close the hole between the two halves; in other words drop the low sign and get the 70-bit magnitude into AR1–35, MQ0–34.

If the IDIV dividend in AR is negative, negate it and set the negative dividend flag. Move the one word dividend in positive form to MQ and clear AR. Shift MQ left, as the algorithm operates on a double length dividend in both types of division although the high part is null in this case.

After the dividend is set up, compare the divisor with it to determine whether the division can be performed. Subtract the absolute value of the divisor from the high half of the dividend (if the divisor is positive, subtract it; if negative, add it). Since the dividend is positive, the result is also positive if the magnitude of the divisor is less than or equal to the number in AR. For a fixed fraction, the divisor is subtracted from the actual dividend and no overflow is allowed. For a fixed integer, AR is clear and the result is positive only for a zero divisor; the worst possible case is the division of $2^{35} - 1$ by 1, whose integral result can be accommodated. (Placing the one word dividend in MQ effectively multiplies it by $2^{-35}$, making it the fractional part of a two word dividend with the binary point in the middle. The quotient is then a proper fraction, which is multiplied by $2^{35}$ simply by interpreting it as an integer.) Thus if the result of this initial subtraction is positive, set Overflow and No Divide, and terminate the procedure so the processor goes on to the next instruction. Dividing by zero is of course meaningless. The reason for prohibiting a fractional division where the result would be greater than 1 is that it is impossible to determine the position of the binary point in the quotient. So it is up to the programmer to shift the dividend to the correct position beforehand. If the result of the initial subtraction is negative, the division can be performed and the processor goes into the division loop.

In division on paper, one subtracts out the divisor the number of times it goes into the dividend, then shifts the dividend one place to the left (or the divisor to the right) and again subtracts out. In binary computations the divisor goes into the dividend either once or not at all. Each subtraction of the divisor thus generates a single bit of the quotient. If the subtraction leaves a positive difference, *ie* if the dividend is larger than the divisor, a 1 is entered into the quotient. If the difference is negative, a 0 is entered. To compensate for subtracting too much, the hardware could add the divisor back into the dividend before going to the next subtraction step. But the PDP–10 algorithm instead shifts first and adds the divisor back in at the new position. It then continues to shift and add putting 0s into the quotient until the result again becomes positive. This procedure generates the same quotient without ever going back a step.

The hardware procedure is as follows. As each addition or subtraction is formed in the adder, put the result in AR shifted one place to the left with AR35 receiving a new bit of the dividend from MQ0, and shift MQ left bringing in a bit of the quotient at MQ35. The bit brought in is the complement of the sign from the adder: if the divisor does not go into the dividend, the resulting minus sign (1) produces a 0 quotient bit; if the divisor does go in, the plus sign gives a 1. Each step loads one bit of the quotient into MQ35, and the low half of the dividend is shifted out of MQ as the quotient is shifted in.

The first step is the test subtraction. In each subsequent step, subtract the absolute value of the divisor if the quotient bit generated in the previous step is 1, but add it back in if the quotient bit is 0. Since the divisor may have either sign, subtract it algebraically if its sign differs from the quotient bit, add it if its sign is the same.

The hardware executes 36 steps to generate 35 magnitude bits. The initial test step must give a 0, which serves as the sign since we are producing a positive quotient. In the final step put the result of the addition or subtraction directly in AR without shifting so the remainder is in the correct position, but shift MQ left putting the sign from the first step in MQ0 and bringing in the last quotient bit. (The bit dropped out of MQ0 is superfluous; it was brought into MQ35 when the hole was closed between the dividend halves.)

To complete the division we must make sure the remainder is correct and determine the correct signs of the results. Since the operations were performed on positive operands, the remainder should also be positive. A negative remainder indicates that too much has been subtracted. To correct this add the absolute value of the divisor back in. If the negative dividend flag is set, negate AR so the remainder has the sign of the original dividend.

Now move the corrected remainder to MQ and move the quotient to AR. If the negative dividend flag and the divisor sign are of opposite states, negate AR to produce the correct quotient sign. The correct quotient and remainder are now in AR and MQ ready for storage.

As an example of the way this algorithm operates, consider a division of 3-bit fixed fractions with a dividend of +.100100 and a divisor of +.101. By paper computation we obtain the quotient this way.

```
          .111
    101 ⟌100.100
         10 1
         ----
          10 00
           1 01
           ----
            110
            101
            ---
              1
```

Taking the processor registers to be four bits in length, AR contains 0.100, MQ has 0.100, and BR has 0.101. Before starting we close the hole changing MQ to 1.000. The sequence has four steps.

```
             0.100    1.000
            -0.101
             -----
             1.111
    1 ←      1.111    0.000
            +0.101
             -----
             0.100
    2 ←      1.000    0.001
            -0.101
             -----
             0.011
    3 ←      0.110    0.011
            -0.101
             -----
             0.001
    4        0.001   ←0.111
```

The quotient is in MQ at the right, the remainder in AR at the left.

## FLOATING POINT ALGORITHMS

§1.1 explains floating point numbers and §2.6 discusses the general characteristics of floating point arithmetic. Exponent computations are done in the SC adder using the exponents and signs from the floating point operands. Remember, the sign is that of the whole number, not of the exponent. Although bits 1–8 of a floating point number represent an exponent in the range −128 to +127, the discussion is entirely in terms of the excess 128 exponents in positive form, *ie* the set of numbers 0–255. Computations generally use twos complement operations even though the exponent in a negative number is a ones complement. The SC sign bit is used to detect exponent overflow and underflow.

After exponent calculations are complete, operations on the fractions are done by the fixed point logic in AR, BR and MQ. Bits 1–8 of AR and BR are filled with null bits, 0s in a positive number, 1s in a negative. Double length operands are in AR and MQ with MQ8–35 forming a magnitude extension of AR. In almost all circumstances the logic treats AR0–35 and MQ8–35 as a single 64-bit register; in all two-word shifting AR35 is connected to MQ8 and MQ0–7 is ignored. Except in division the fixed point calculation generates a double length fraction, which is shifted arithmetically (in right shifting the sign goes into AR1; in left shifting the sign is unaffected and 0s enter MQ35). Almost all floating point instructions normalize the result, thus making use of the low order part even though the instruction may store only the high order word.

**Addition, Subtraction.** $E$,0 or the word from location $E$ is in BR, and AC is in AR. For subtraction move the negative of the subtrahend from BR to AR and move the minuend from AR to BR. This reduces subtraction to addition, so the rest of the algorithm is the same for both.

The initial objective is to determine the difference between the exponents and to determine which exponent is the larger. If the signs of the operands differ, add the exponents into SC. If the signs are the same, subtract the BR exponent from the AR exponent by adding the twos complement. Let $x$ and $y$ be the AR and BR exponents in positive form. The table below shows the calculations as a function of the operand signs, and the sign of the result in SC as a function both of the operand signs and the relative values of $x$ and $y$.

| AR+, BR+ | | AR+, BR− | | AR−, BR+ | | AR−, BR− | |
|---|---|---|---|---|---|---|---|
| $+[x]$ | | $+[x]$ | | $-[255-x]$ | | $-[255-x]$ | |
| $-[256-y]$ | | $-[255-y]$ | | $+[y]$ | | $+[1+y]$ | |
| $-[256+x-y]$ | | $-[255+x-y]$ | | $-[255-x+y]$ | | $-[256-x+y]$ | |
| SC+ | SC− | SC+ | SC− | SC+ | SC− | SC+ | SC− |
| $x \geqslant y$ | $x < y$ | $x > y$ | $x \leqslant y$ | $x < y$ | $x \geqslant y$ | $x \leqslant y$ | $x > y$ |

As can be seen from the above, if AR already contains the number with the smaller exponent, the SC and AR signs differ. Hence if the SC and AR signs are the same, switch BR and AR so the number with the smaller exponent can be shifted. If the exponents are equal, the signs may or may not be the same but it matters not whether the transfer takes place.

To control the shifting we must now get the negative of the difference between the exponents. Let $d$ be $|x-y|$. There are four cases as a function of the SC sign and whether the AR and BR signs are equal. The second column lists the present contents of SC, the third tells what must be done to arrive at $-[256-d]$ in SC.

| | | |
|---|---|---|
| SC+, AR0 = BR0 | $+[d]$ | Negate SC |
| SC+, AR0 ≠ BR0 | $+[d-1]$ | Complement SC |

| | | |
|---|---|---|
| SC−, AR0 = BR0 | $-[256-d]$ | Do nothing |
| SC−, AR0 ≠ BR0 | $-[255-d]$ | Add 1 to SC |

If $d < 64$ (indicated by a negative SC with a 0 in either SC1 or SC2) nullify AR1−8 and shift AR and MQ right $d$ places so its bits correctly match the BR bits in order of magnitude. If $d > 64$ clear AR for its contents are of no significance.

Now move the larger exponent from BR to SC in positive form, nullify BR1−8, and add BR and AR into AR as fixed fractions. Finally enter the normalizing sequence.

This sequence first tests for a zero result. If AR and MQ8−35 are clear, bypass the rest of the procedure. If the fractional result has overflowed into AR8 (indicated by AR0 ≠ AR8 or AR8 = 1 and AR9−35 = 0), shift right and increase the exponent by one. The number is now normalized.

Complement the exponent in SC. If the instruction is not UFA and the number is not normalized go into the normalizing loop. In each step shift the double length fraction left and add 1 to the negative exponent (decreasing its magnitude by 1). Terminate the loop when the fraction is normalized, indicated by the sign and the MSB of the fraction being different (AR0 ≠ AR9) or the magnitude being ½ (AR9 = 1 and AR10−35 = 0).

If the instruction specifies rounding, adjust the high fraction so it is rounded and is in twos complement form if negative. The rounding is away from zero. For a positive result the high fraction must be increased if the low fraction is greater than half the value of the high fraction LSB. In a negative result the high fraction is a ones complement, which is one greater in magnitude than the twos complement. Hence it is already rounded and should be decreased in magnitude if the low fraction is < ½LSB. In either case add $2^{-27}$ into AR if MQ8 is 1 unless MQ9−35 is clear in a negative number. A 1 in MQ8 indicates a low fraction ⩾ ½LSB in a positive number, ⩽ ½LSB in a negative number. The condition that MQ9−35 not be zero in a negative number is the case where the low fraction is exactly ½LSB. If the high fraction is actually changed, renormalize it. A single normalizing shift is all that is required and it occurs in only two cases: a right shift when $1-2^{-27}$ is rounded, a left shift when $-½$ is changed to a correct twos complement.

Once the number has been normalized (and rounded if necessary) the exponent is in negative form. Thus if the SC sign bit is 0, set Overflow and Floating Overflow. If SC1 is also 0, the sign bit must have been changed by decreasing the exponent, so also set Floating Underflow (the maximum possible exponent overflow is 128 giving an SC contents of $777_8$, and this can occur only in division). Insert the exponent in correct form into AR1−8.

The result is now ready to store from AR unless the instruction is in long mode. To ready the double length result subtract 27 from the positive exponent in SC. Save the high word in MQ, and move the low word to AR, but only if the decreased exponent is still positive. If the sign is 1, the true exponent of the low word is less than −128, so clear AR. (Note that this condition is also true if the low exponent is > 127, which can occur only if the high exponent is > 154.) If the low word is nonzero, shift AR right one place to put the fraction in bits 9−35 (remember that all shift operations use MQ8–35), clear AR0 so the low word has a positive sign even if the double length fraction is negative, and insert the low exponent in positive form in bits 1–8. Finally switch AR and MQ so the high and low words are in correct position for storage.

**Scaling.** The 9-bit signed scale factor from bits 18 and 28–35 of $E$ is in SC, and AC is in AR and BR. If the floating point number being scaled is positive, simply add the sign and exponent from BR0–8 to SC; if the number is negative, add the complement of BR0–8 to SC. Let $x$ be the exponent in positive form and let $y$ be the absolute value of the scale factor. There are only two cases,

$$\begin{array}{l} +[x] \\ \underline{+[y]} \\ +[x+y] \end{array} \qquad \begin{array}{l} +[x] \\ \underline{-[256-y]} \\ +[x-y] \end{array}$$

and in either the result is in positive form in SC.

Now enter the normalizing sequence described under floating addition. Only left shifting can occur bringing 0s in from MQ. The result can be zero, and exponent overflow or underflow can occur; but there is no rounding, and at the end the one-word result is in AR ready for storage.

**Multiplication.** $E$,0 or the word from location $E$ is in BR, and AC is in AR. Place the AR exponent in positive form in SC, and add the positive form of the BR exponent to it. Since both are in excess-128 code, subtract 128. Save the result in the floating exponent register FE so SC can be used to control the multiplication of the fractions.

Nullify the exponent parts of AR and BR. Move the multiplier from BR to MQ and the multiplicand from AR to BR. Clear AR. Now multiply the fractions by the same procedure given for fixed point multiplication with the following differences:

◆ There are only 28 steps instead of 36.

◆ The shift register extension of AR for the construction of the product is MQ8–35. As the multiplier is shifted out, bits of the product come in at MQ8.

◆ In the final step place the adder output directly into AR but do not shift MQ – the low fraction is in MQ8–34, the correct position for normalization.

Clear MQ35, move the exponent back to SC, and enter the normalizing sequence described under floating addition. If the operands are normalized, at most one left shift is needed to normalize the result.

**Division.** The divisor, $E$,0 or the contents of location $E$, is in BR. The dividend from AC is in AR. In long mode the low half of the dividend from the second accumulator is in MQ; otherwise MQ is clear.

If the dividend is negative, make it positive and set the negative dividend flag. Except in long mode, negate the dividend simply by negating AR. For long mode follow the procedure given for DIV in the second paragraph of the fixed division algorithm. With a floating point operand the left MQ shift puts the low fraction in MQ8–34.

Place the AR exponent in positive form in SC. Subtract the magnitude of the BR exponent from it by adding the negative form of the exponent (ones complement) plus 1. Since the excess-128 factors cancel in the subtraction, add 128. Save the result in the floating exponent register FE so SC can be used to control the division of the fractions.

Nullify the exponent parts of AR and BR. Subtract the absolute value of the divisor from the high half of the dividend. If the result is positive, indicating the divisor is less than or equal to the dividend, shift AR and MQ right and increase the exponent in SC by 1. Save the adjusted exponent in FE. The shift divides by 2, so if the operands are normalized, the dividend must now be less than the divisor.

Now divide the fractions by the same procedure given for fixed point fractional division with the following differences:

◆ Since the dividend has already been adjusted, the test in the first step stops the division only if the divisor is zero, or is unnormalized and less than the dividend. A normalized divisor cannot cause the quotient to overflow. If the result of the initial subtraction is positive, terminate the procedure and set Floating Overflow as well as Overflow and No Divide.

◆ Instead of 36 steps there are only 29 if the instruction specifies rounding, otherwise 28.

◆ The shift register extension of AR is MQ8–35. As quotient bits are brought in at MQ35, dividend bits are supplied to AR35 from MQ8. The shifting clears MQ0–7.

◆ The MQ shift in the final step places a 27-bit quotient fraction in MQ9–35 or a 28-bit fraction in MQ8–35.

◆ As in the fixed point algorithm generate the correct signed remainder, put it in MQ, and move the quotient to AR but leave it positive.

If the instruction specifies rounding, shift AR right placing the 27-bit fraction in the correct position, and if the bit shifted out of AR35 is 1, add it back into AR35 to round the positive quotient. If the quotient is zero bypass the rest of the procedure. The reaminder will also be zero except in an FDVL where the double length dividend is unnormalized and its high fraction is zero.

Complement the exponent in SC. If the instruction uses normalized operands the initial dividend adjustment guarantees that the quotient will be normalized. If it is not, shift AR left (bringing 0s into AR35) until a 1 appears in AR9, each time increasing the negative exponent by 1 (decreasing its magnitude).

Since the exponent is in negative form, if SC0 is 0, set Overflow and Floating Overflow. If SC1 is also 0, the sign bit must have been changed by decreasing the exponent, so also set Floating Underflow. Insert the exponent in correct form into AR1–8. If the negative dividend flag and the divisor sign (BR0) are of opposite states, negate AR to produce the correct quotient sign.

The quotient is now ready for storage from AR and the remaining operations are performed only for long mode. Save the quotient in BR and bring the high half of the original dividend from AC to AR. Put the dividend exponent in SC. Decrease its magnitude by 26 if the dividend was shifted right at the beginning to allow the division to be performed; otherwise decrease it by 27. Move the remainder to AR and insert the exponent in it provided the remainder is not zero and the exponent is within the proper range, −128 to 127 (the test is that the sign resulting from the exponent calculation is the same as the sign of the remainder). If the exponent is outside that range clear AR; the assumption is that the remainder is of no significance (*ie* the exponent is too small). Move the remainder with its correct exponent from AR to MQ and put the quotient back in AR. The two words are now ready for storage.

**Double Precision Division.** The software routine that performs double precision floating point division and the algorithm it utilizes are given at the end of §2.11. FDVL performs the division

$$A/b \quad = \quad q + r2^{-27}/b$$

where $q$ and $r$ are the quotient and remainder. In a double precision division the divisor is of the form

$$B \quad = \quad b + d2^{-27}$$

Using the expansion

$$\frac{1}{x+y} \quad = \quad \frac{1}{x}\left[1 - \frac{y}{x} + \frac{y^2}{x^2} - \frac{y^3}{x^3} + \ldots\right] \qquad (y^2 < x^2)$$

and letting $x = b$ and $y = d2^{-27}$ gives

$$\frac{A}{B} \quad = \quad \left(q + \frac{r2^{-27}}{b}\right)\left[1 - \frac{d2^{-27}}{b} + \frac{d^2 2^{-54}}{b^2} - \frac{d^3 2^{-81}}{b^3} + \ldots\right]$$

Multiplying out and gathering like terms gives

$$\frac{A}{B} \quad = \quad q + \frac{1}{b}(r - qd)2^{-27} - \frac{d}{b^2}(r - qd)2^{-54} + \frac{d^2}{b^3}(r - qd)2^{-81} - \ldots$$

where the first two terms on the right are those in the equation at the bottom of page 2-67.

The ratio of adjacent terms is

$$\frac{T_{n+1}}{T_n} \quad = \quad \frac{-d2^{-27}}{b}$$

In an alternating convergent series, the error due to truncation is smaller than the first term dropped. Therefore

$$|Error| \quad < \quad \frac{d2^{-27}}{b} T_n$$

Since the maximum value of $d$ is less than 1 and the minimum value of $b$ (normalized) is ½,

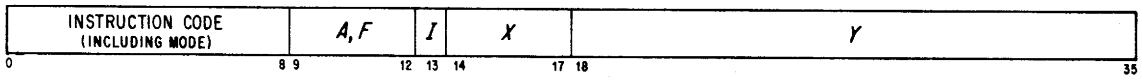
$$|Error| \quad < \quad T_n 2^{-26}$$

# APPENDIX G

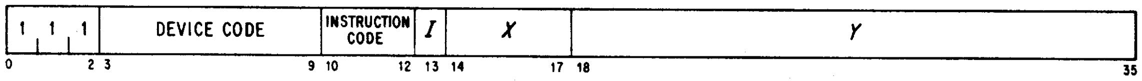
## BIT ASSIGNMENTS

The drawing on pages G2 and G3 shows the formats of the various types of words used by the processor. Bit assignments in the condition and data words for the IO instructions will be added later.

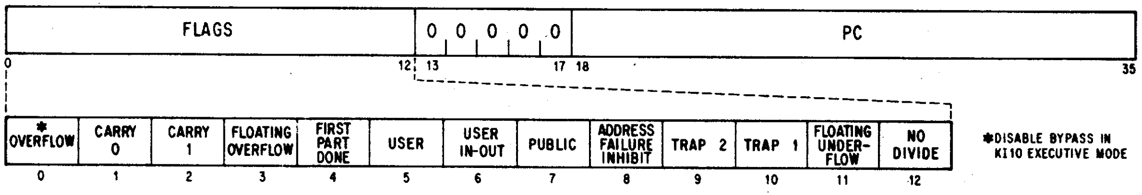
## BASIC INSTRUCTIONS



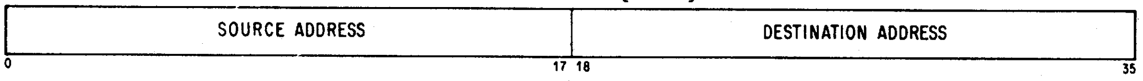
## IN-OUT INSTRUCTIONS



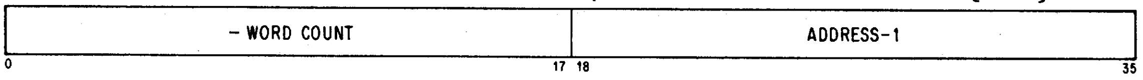
## PC WORD



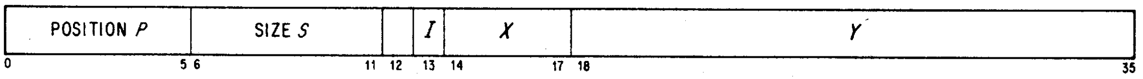
## BLT POINTER {XWD}



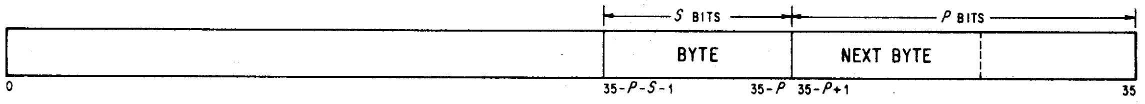
## BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD {IOWD}



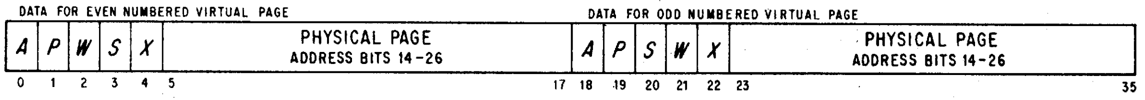
## BYTE POINTER



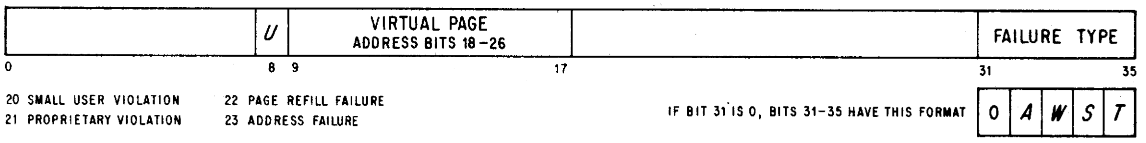
## BYTE STORAGE



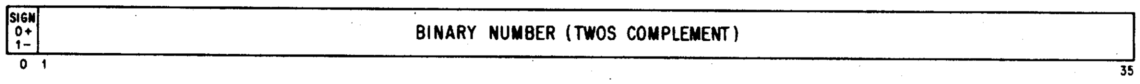
## PAGE MAP WORD


DATA FOR EVEN NUMBERED VIRTUAL PAGE
DATA FOR ODD NUMBERED VIRTUAL PAGE
A P W S X
PHYSICAL PAGE ADDRESS BITS 14–26
A P S W X
PHYSICAL PAGE ADDRESS BITS 14–26
0 1 2 3 4 5 17 18 19 20 21 22 23 35

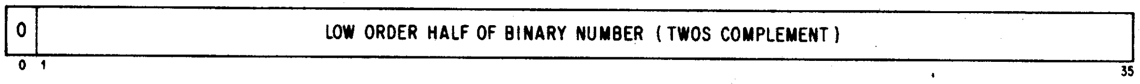
## PAGE FAIL WORD



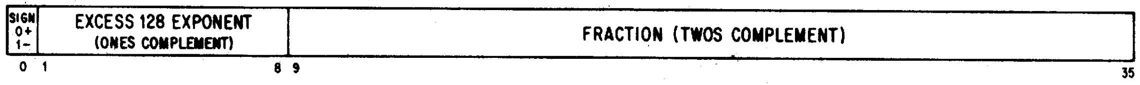
FIXED POINT OPERANDS



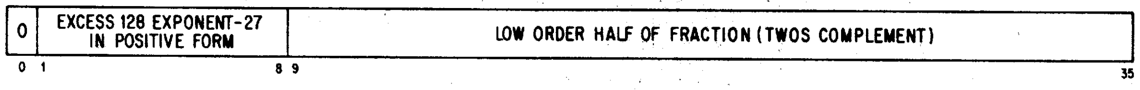
LOW ORDER WORD IN DOUBLE LENGTH FIXED POINT OPERANDS



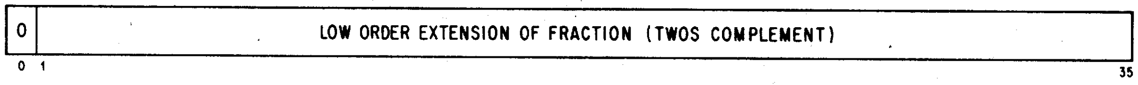
FLOATING POINT OPERANDS



LOW ORDER WORD IN SOFTWARE DOUBLE LENGTH FLOATING POINT OPERANDS



LOW ORDER WORD IN HARDWARE DOUBLE LENGTH FLOATING POINT OPERANDS



## 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 |

## IN-OUT DEVICE BIT ASSIGNMENTS

| DEVICE | CODE | FUNCTION | Ø/18 | 1/19 | 2/20 | 3/21 | 4/22 | 5/23 | 6/24 | 7/25 | 8/26 | 9/27 | 1Ø/28 | 11/29 | 12/3Ø | 13/31 | 14/32 | 15/33 | 16/34 | 17/35 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APR | ØØØ | CONO | CLR PDL OV | I/O RESET | | CLR ADR BREAK | CLR MEM PROT | CLR NXM | CLR CLOCK ENABLE | SET CLOCK ENABLE | CLR CLOCK FLAG | CLR FOV ENABLE | SET FOV ENABLE | CLR FOV FLAG | CLR AROV ENABLE | SET AROV ENABLE | CLR AROV FLAG | PIA | | |
| | | CONI | | PDL OV FLAG | IOT USER FLAG | ADR BREAK FLAG | MEM PROT FLAG | NXM FLAG | | CLOCK ENABLE | CLOCK FLAG | | FOV ENABLE | FOV FLAG | TRAP OFFSET | AROV ENABLE | AROV FLAG | PIA | | |
| | | DATAO | PROTECTION REGISTER (LH Ø-7) RELOCATION REGISTER (RH 18-25) | | | | | | | | | | | | | | | | | |
| | | DATAI | ← 36 DATA SWITCHES → | | | | | | | | | | | | | | | | | |
| PI | ØØ4 | CONO | CLEAR POWER FAIL FLAG | CLEAR PARITY FLAG | CLEAR PARITY ENABLE | SET PARITY ENABLE | | CLEAR PI SYSTEM | REQUEST INTERRUPT ON CHANNEL | TURN ON CHANNELS | TURN OFF CHANNELS | TURN OFF PI ACTIVE | TURN ON PI ACTIVE | SELECT CHANNELS 1-7 FOR BITS 24, 25, 26 | | | | | | |
| | | CONI | POWER FAILURE FLAG | PARITY ERROR FLAG | PARITY ENABLE | INTERRUPT IN PROGRESS ON CHANNELS 1-7 | | | | | | | PI ACTIVE | CHANNEL ACTIVE 1-7 | | | | | | |
| | | DATAO | ← 36 BITS TO THE MEMORY INDICATORS → | | | | | | | | | | | | | | | | | |
| ADC A-D CONVERTER (AD1Ø) | Ø24 | | | | | | | | | | | | | | | | | | | |
| PTP | 1ØØ | CONO | | | | | | | | | | | | | BINARY | BUSY | DONE FLAG | PIA | | |
| | | CONI | | | | | | | | | | | | OUT OF TAPE | BINARY | BUSY | DONE FLAG | PIA | | |
| | | DATAO | | | | | | | | | | | HOLE 8 UNLESS BINARY | HOLE 7 UNLESS BINARY | HOLE 6 | HOLE 5 | HOLE 4 | HOLE 3 | HOLE 2 | HOLE 1 |
| PTR | 1Ø4 | CONO | | | | | | | | | | | | | BINARY | BUSY | DONE FLAG | PIA | | |
| | | CONI | | | | | | | | | | TAPE FLAG | | | BINARY | BUSY | DONE FLAG | PIA | | |
| | | DATAI | ← 36 BIT WORD IF BINARY, 8 BITS (28-35) IF NOT BINARY → | | | | | | | | | | | | | | | | | |
| TTY | 12Ø | CONO | | | | | | | TEST FLAG | TTI BUSY CLEAR | TTI FLAG CLEAR | TTO BUSY CLEAR | TTO FLAG CLEAR | TTI BUSY SET | TTI FLAG SET | TTO BUSY SET | TTO FLAG SET | PIA | | |
| | | CONI | | | | | | | TEST FLAG | | | | | TTI BUSY | TTI FLAG | TTO BUSY | TTO FLAG | PIA | | |
| | | DATAO | | | | | | | | | | | 8 BIT CHARACTER TO TELETYPE | | | | | | | |
| | | DATAI | | | | | | | | | | | 8 BIT CHARACTER FROM KEYBOARD | | | | | | | |
| LPT (LP1Ø) | 124 | CONO | | | | | | | | CLEAR PRINTER | | | BUSY | DONE FLAG | ERROR PIA | | | DONE PIA | | |
| | | CONI | | | | | | | 128 CHAR | 96 CHAR | | ERROR | BUSY | DONE FLAG | ERROR PIA | | | DONE PIA | | |
| | | DATAO | FIRST CHARACTER | | | | | | | SECOND CHARACTER | | | | | | | THIRD | | | |
| | | | CHARACTER | | | FOURTH CHARACTER | | | | | | | FIFTH CHARACTER | | | | | | | |
| PLT (XY1Ø) | 14Ø | CONO | | | | | | | | | | | | | | BUSY | DONE FLAG | PIA | | |
| | | CONI | | | | | | | | | | | | | POWER ON | BUSY | DONE FLAG | PIA | | |
| | | DATAO | | | | | | | | | | | | | PEN RAISE | PEN LOWER | -X DRUM UP | +X DRUM DOWN | +Y CARRIAGE LEFT | -Y CARRIAGE RIGHT |
| CR (CR1Ø) | 15Ø | CONO | | | | | | CLEAR READER | OFFSET CARD | | READ CARD | TROUBLE ENABLE | CLEAR DATA MISSED | READY ENABLE | CLEAR END OF FILE | CLEAR END OF CARD | CLEAR DATA FLAG | PIA | | |
| | | CONI | READER TRBLE ENAB | READER READY ENAB | PICK ERROR | PHOTO CELL ERROR | CARD MOTION ERROR | READER STOP | CARD IN READER | HOPPER STACKER | READING CARD | READER TROUBLE | DATA MISSED | READY TO READ | END OF FILE | END OF CARD | DATA FLAG | PIA | | |
| | | DATAI | | | | | | | ROW 12 | ROW 11 | ROW Ø | ROW 1 | ROW 2 | ROW 3 | ROW 4 | ROW 5 | ROW 6 | ROW 7 | ROW 8 | ROW 9 |

THIS IS A TEMPORARY PAGE

| DEVICE | CODE | FUNCTION | Ø/18 | 1/19 | 2/2Ø | 3/21 | 4/22 | 5/23 | 6/24 | 7/25 | 8/26 | 9/27 | 1Ø/28 | 11/29 | 12/3Ø | 13/31 | 14/32 | 15/33 | 16/34 | 17/35 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DSK (RC1Ø) | 17Ø | CONO | SELECT SECTOR CTR | | CLEAR DISK DESIG ERROR | CLR TRACK SELECT ERROR | CLEAR DISK NOT READY | CLEAR PWR SUP FAILURE | CLR DISK PAR ERR | CLR CHAN DATA PAR ERR | CLR CHAN CONTROL PAR ERR | CLEAR NXM | CLEAR ILLEGAL WRITE | CLEAR OVER-RUN | WRITE CHAN CNTL WD | STOP CLEAR BUSY | CLEAR DONE | PIA | | |
| | | CONI LH | | | | | | MAINT SECTOR SELECT | PROTECT LOWER | ← BOUNDARY SWITCHES (BCD) → DISK | | TRACK 1ØØ'S | 1Ø'S | | | | 1'S | | | |
| | | CONI RH | DATA XFER IN PROG | SEARCH ERROR | DISK DESIG ERROR | TRACK SELECT ERROR | DISK NOT READY | PWR SUPPLY FAIL | DISK PARITY ERROR | CHANNEL DATA PAR ERR | CHANNEL CONTROL PAR ERR | NXM | ILLEGAL WRITE | OVER-RUN | CHAN CNTL WD WRITTEN | BUSY | DONE | PIA | | |
| | | DATAO LH | DISK SELECT | | ← TRACK (BCD) → 1ØØ'S | 1Ø'S | | | | 1'S | | | | ← SECTOR (BCD) → 1Ø'S | | | 1'S | | | |
| | | DATAO RH | INITIAL PARITY | | | | | | DISABLE PAR ERR STOP | DISABLE MEMORY PAR ERR STOP | WRITE | INITIAL CHANNEL CONTROL WORD ADDRESS | | | | | | | | WRITE EVEN PARITY |
| | | DATAI RH | PARITY REGISTER | | | | | | | | SECTOR CTR SELECTED | | ← SECTOR COUNTER (BCD) → 1Ø'S | | | | 1'S | | | |
| DTC (TD1Ø) | 32Ø | CONO | STOP | GO FORWARD | GO REVERSE | DELAY INHIBIT | SELECT | CLEAR SELECT & UNIT Nº | TRANSPORT NUMBER | | | FUNCTION NUMBER Ø = NOTHING 4 = WRITE MK & TM | | | DATA PIA | | | FLAGS PIA | | |
| | | CONI | STOP | GO FORWARD | GO REVERSE | | SELECT | DESELECT | TRANSPORT NUMBER | | | 1 = READ ALL 5 = WRITE ALL 2 = READ BLOCK # 6 = WRITE BLOCK # 3 = READ DATA 7 = WRITE DATA | | | DATA PIA | | | FLAGS PIA | | |
| | | DATAO | ← 36 BIT WORD → | | | | | | | | | | | | | | | | | |
| | | DATAI | ← 36 BIT WORD → | | | | | | | | | | | | | | | | | |
| DTS | 324 | CONO | PARITY ERR ENABLE | DATA MISSED ENABLE | JOB DONE ENABLE | ILLEGAL OPERATION ENABLE | END ZONE ENABLE | BLOCK MISSED ENABLE | | | | | | | | | | | STOP ALL TRANSPORTS | FUNCTION STOP |
| | | CONI LH | PARITY ERR ENABLE | DATA MISSED ENABLE | JOB DONE ENABLE | ILLEGAL OPERATION ENABLE | END ZONE ENABLE | BLOCK MISSED ENABLE | DELAY IN PROGRESS | ACTIVE | UP TO SPEED | BLOCK NUMBER | REVERSE CHECK | DATA | FINAL DATA | CHECKSUM | IDLE | BLOCK NUMBER READ | | FUNCTION STOP |
| | | CONI RH | PARITY ERROR | DATA MISSED | JOB DONE | ILLEGAL OPERATION | END ZONE | BLOCK MISSED | WRITE LOCK ON | WRTM SWITCH ON | INCOMPLETE BLOCK | | MARK TRACK ERROR | SELECT ERROR | | | | | FLAG PI REQUEST | DATA REQUEST |
| | | DATAO RH | | | | UP TO SPEED TEST | | | | | | | | | | | | | | MARK TRACK DATA |
| | | DATAI LH | | | | | | | | | | | | | | | | PARITY | | |
| | | DATAI RH | REGISTER | | | READ/WRITE REGISTER | | | | | | | | NOT MK BN SPACE | NOT MK BN END | NOT MK DATA SYNC | NOT MK REV DATA END | NOT MK DATA | NOT MK FWD DATA END | NOT MK BN SYNC |
| DLS (DC1Ø) | 24Ø | CONO | | | | | | | | | | | | | CLEAR DLS | SET DATA TERM RDY | RESET SCANNER | PIA | | |
| | | CONI | | | | | | | | | | | | | DTR DISABLED | TRANS FLAG | RCVR FLAG | PIA | | |
| DC1ØB/ DC1ØE | | DATAO LH | | | | | | | | | | | | USE LINE NO. | LINE NUMBER | | | | | |
| | | DATAO RH | | | | | | | | | | TRANS DISAB | BIT 8 CAUSE PI | BIT 7 OFF HOOK EN | BIT 6 CRQ | BIT 5 DPR | BIT 4 NB8 | BIT 3 NB4 | BIT 2 NB2 | BIT 1 NB1 |
| | | DATAI LH | | | | | | | | | | | | | LINE NUMBER | | | | | |
| | | DATAI RH | | | | | | | | | | RCVR DATA | BIT 8 | BIT 7 OFF HOOK EN | BIT 6 DLO | BIT 5 PND | BIT 4 ACR | BIT 3 CB | BIT 2 CE | BIT 1 |
| TMC (TM1Ø) | 34Ø | CONO | UNIT | | | PARITY Ø = EVEN 1 = ODD | CORE DUMP | FUNCTION Ø = NO-OP 1 = REWIND 2 = READ 3 = RD COMPARE 4 = WRITE 5 = WRITE EOF 6 = SPACE FWD 7 = SPACE REV 1Ø = NO-OP, PI INT ON TUR 12 = READ ACRS RECORD 13 = RD/CMP ACRS RECD 14 = WR W/LONG EOR 16 = SPACE FILE 17 = SPACE FILE REV | | | | NEXT UNIT ENAB | DENSITY Ø = 2ØØ 1 = 556 2 = 8ØØ 3 = (8ØØ) | | FLAG PIA | | | DATA PIA ‡ | | |
| | | CONI RH | UNIT | | | PARITY | CORE DUMP | | | | | NEXT UNIT ENAB | DENSITY | | FLAG PIA | | | DATA PIA | | |
| | | CONI LH | | | | | | | | | | | | | | | | NEXT UNIT | | |
| | | DATAO 7-CHN OR CORE DMP | ← 36 BIT WORD → (7 CHN: SIX 6 BIT CHARACTERS AS IN DATAI BELOW) | | | | | | | | | | | | | | | | | |
| | | DATAO 9-CHN | ← 1 ST CHARACTER → | | | | | | | | ← 2ND CHARACTER → | | | | | | | | ← | |
| | | | ← 3 RD CHARACTER → | | | | | | ← 4 TH CHARACTER → | | | | | | | | IGNORED | | | |
| | | DATAI 7-CHN | ← 1 ST CHARACTER → | | | | | | ← 2 ND CHARACTER → | | | | | | ← 3 RD CHARACTER → | | | | | |
| | | | ← 4 TH CHARACTER → | | | | | | ← 5 TH CHARACTER → | | | | | | ← 6 TH CHARACTER → | | | | | |
| ‡ BITS FOR TM1ØA ONLY | | DATAI 9-CHN | ← 1 ST CHARACTER → | | | | | | | | ← 2ND CHARACTER → | | | | | | | | ← | |
| | | | 3 RD CHARACTER → | | | | | | ← 4 TH CHARACTER → | | | | | | | | PAR ERR 1ST CHAR | PAR ERR 2ND CHAR | PAR ERR 3RD CHAR | PAR ERR 4TH CHAR |
| * BITS FOR TM1ØB ONLY | | DATAI 9-CHN CORE DMP | ← 1 ST CHARACTER → | | | | | | | | ← 2ND CHARACTER → | | | | | | | | ← | |
| | | | 3 RD CHARACTER → | | | | | | ← 4 TH CHARACTER → | | | | | | ← 5 TH CHARACTER → | | | | | |
| TMS | 344 | CONO | | | | | | | | | | | | | | CLEAR DATA PAR ERR * | CLEAR WT CW DONE * | WRITE CTRL WORD * | XFER CRC ‡ | STOP |
| | | CONI LH | | | | | | | | | | | | CW PAR ERR * | NON-EX MEM * | DATA PAR ERR * | WT CW DONE * | CHARACTER COUNTER | | |
| | | CONI RH | X PORT HUNG | REWIND-ING | LOAD POINT | ILLEG OP | PAR ERROR | END OF FILE | END OF TAPE | READ COMPARE ERROR | RECORD LENGTH INCORRECT | DATA LATE | BAD TAPE | JOB DONE | TAPE UNIT RDY (TUR) | CHANNEL ERROR * | WRITE LOCK | 7-CHAN TAPE | NEXT UNIT | DATA REQ ‡ |
| | | DATAO RH | | | | | | | | | | INITIAL CHANNEL CONTROL WORD ADDRESS | | | | | | | | WRITE EVEN PARITY |
| CCI (DA1Ø) | Ø14 | CONO | | | | | | CLR SELF TEST | SET SELF TEST | CLR FM1Ø FULL | SET FM1Ø FULL | CLR FM1Ø EMPTY | SET FM1Ø EMPTY | CLR TO 1Ø FULL | SET TO 1Ø FULL | CLR TO 1Ø EMPTY | SET TO 1Ø EMPTY | PIA | | |
| | | CONI | | | | | | | SELF TEST (1) | | FM 1Ø FULL (1) | | FM 1Ø EMPTY (1) | | TO 1Ø FULL (1) | | TO 1Ø EMPTY (1) | PIA | | |
| | | DATAI DATAO FOR PDP8 | 1ST PDP 8 BYTE (A) | | | | | | 2ND PDP 8 BYTE (B) | | | | | | 3RD PDP 8 BYTE (C) | | | | | |
| | | | ← 36 BITS → | | | | | | | | | | | | | | | | | |
| | | DATAI DATAO FOR PDP9 | ← 36 BITS → | | | | | | | | | | | | | | | | | |
| | | | 1ST PDP 9 BYTE (A) | | | | | | | | | 2ND PDP 9 BYTE (B) | | | | | | | | |